Eternity Service Project

Pavel Hlousek, Marek Janata, Jan Pechanec, Vaclav Petricek, Ivana Reisingerova, Jiri Vaculik

Supervisor: Antonin Benes

KSI MFF UK Prague
Charles University Prague
petricek@acm.org

This documentation is an abridged version of the Czech doc. It contains the important parts (ie. generic chapters on security is not included) of our documentation. Sorry for our English and we hope that this paper will be useful for you.

---

7. Index

Access Certificate (AC)
An access certificate to any machine in the Eternity Service. It includes an address or a generalized address (so called onion), Public Key of the server, expire time of the ceritficate and other voluntary items.

ACID
In the Service unambiguous identifier of Access Certificate.

Acs (Access Certificate Server)
A server, which publishes Access Certificates of other servers in the Service.

Bank
A program managing payments and payment's transactions in a financial institution geared to the Eternity Service.

Onion
Generalized address of a machine in it's Access Certificate. Onion includes ciphered path to the given server.

Cipherer
An object encapsulating RSA cryptographical algorithms.

Denial Of Service Attack
An attack, which obstructs access to a service.

DES
A symetric cryptographical algorithm (Data Encryption Standard).

Eso (Eternity Server)
A server offering main services in the Eternity Service - storing and searching a file.

Eternity Service (ES)
A service offering a safe way in storing and searching a file and preventing the file against modification or delete.

GMessage (Generic Message)
An auxiliary class offering an easy way to work with messages.

Client
A program for user, which wants to store or search a data in the Service.

MAC (Message Authentification Code)
A checksum over data, used for prooving ownership of the file.

Majordomo
A main thread of a server managing incomming messages.

Middle Attack
Going through the Service a data are in "the middle" of a path encrypted only with Public Key of the recipient. That is why it is easy to distinguish information about message ID, number of block in the message etc.

Mix
Something like a remailer. A server used for "reposting" of messages, generating padding and an onion.

O-Authentification (OAuth)
A secret key, known only to a bank and a client. It is used for computing MAC.

Onion-routing
A message, which goes through machines in a network, is gradually uncovered slice after slice. Every slice hides information about IP address of a next machine only. This computer has the key to discover next recipient etc. This way each computer knows its predecessor and successor only, but does not know real depositor or recipient.

Padding
Filling incomming messages, which were manipulated, to its original size (before sending them).

PKCS
RSADSI set of cryptographical standards.

Payment Plan
A plan of payments, which is sent by an Eso through a Client to a Bank.

Receiver
An object operating an input socket.

RSA
Asymetric encryption algorithm.

RSADSI
RSA Data Security Inc.

RSAEuro
A cryptographical library, compatibile with RSARef-toolkit.

RSARef
RSADSI cryptographical library, based on PKCS standards.

S-Authentification (SAuth)
A secret key, known only to a bank and a Eso. It is used for identification of the Eso.

Sealed S-Authentification (SealedSAuth, SealSA)
An S-Authentification encrypted with Bank's Public Key.

Sender
An object operating an output socket.

Six
A general service over a Mix.

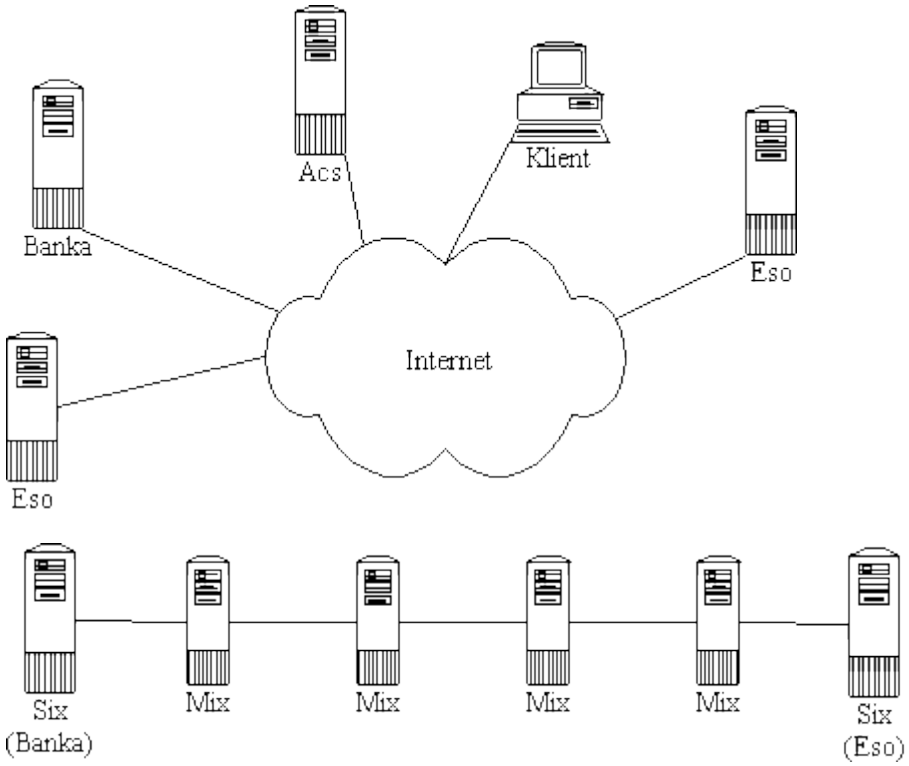
TCB (Trusted Computing Base)
A cryptographical device, which has its Secret Key stored in hardware, and which can realize simple cryptographical algorithms. It can decrypt data with its Private Key and encrypt them with another key. It is garanted that nobody can see not encrypted plain data outside the TCB.

TCBWrapper
A class encapsulating general behavior of TCB.

Traffic analysis
Watching network traffic for analysis of posted messages.

8.3 Architecture

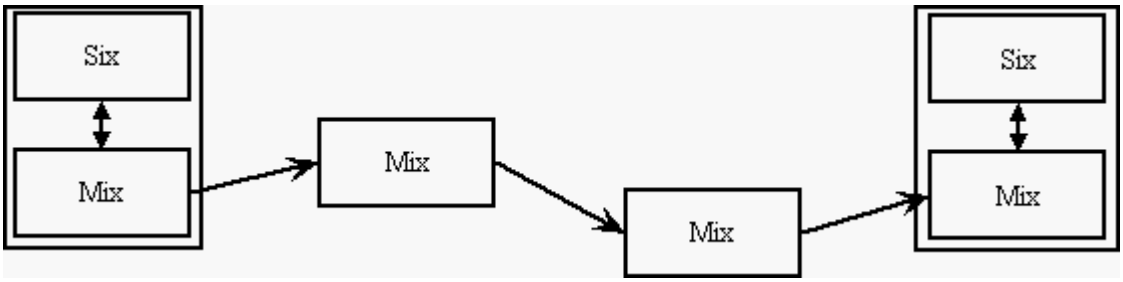
All servers in the Eternity Service have to communicate together in the environment of Internet. An user needs to store his file on a server, which stores data and makes possible its downloading, bank needs to agree with this server about payments etc. From a bird perspective whole Service could look like this:



The servers on the picture would provide whole functionality of the Service. But the problem is, that we have bigger demands on the Service security. If there would be an easy way to find out where are the servers managing stored files, no difficulty should appear when discarding or damaging these servers. This could leed to the collaps of the whole System (see chapter 11.). To prevent from such a situation or reduce chances of incidental attacks on the Eternity Service all communication goes through special servers (Mixes), which make very difficult to trace the communication and mask real executive of the Service. From the view of the formerly mentioned servers Mixes on the path are transparent.

Each executive server tightly cooperates with one Mix. Typically these are two processes runing on one machine or on two machines connected by a controlled

(secure) network. For the server this Mix is a gate to the Eternity Service. On the other hand Mix does not distinguish whichone of the executive servers it communicates with - it is easily service above Mix - so called Six. The following picture shows used architecture.



---

11. System safety

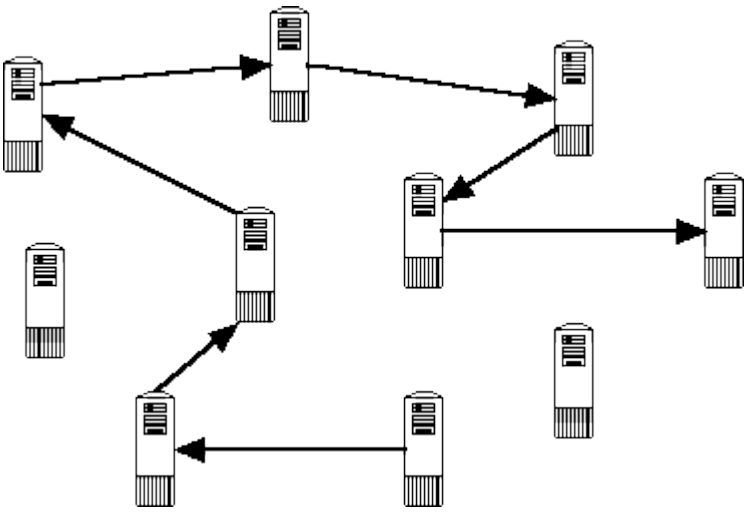
11.1. Onion routing

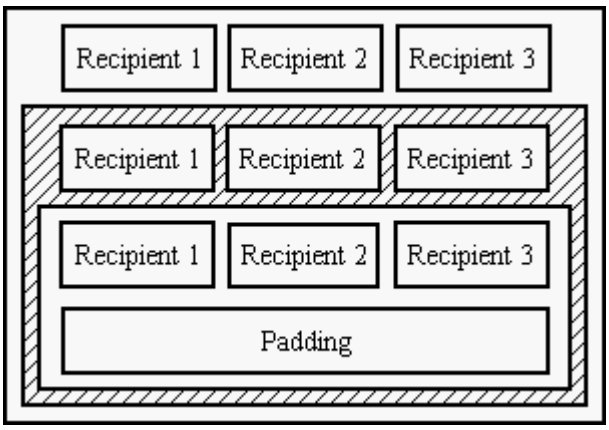
Most of the current systems for safe communication are focused on securing the content of messages using symetric or asymetric keys. These systems take for granted that communicating parties know the exact address of each other. Besides that they also leave apart the question of traffic analysis. For our system, that is trying to hide servers storing data as much as it is possible, reavealing the identity of communicating parties could be fatal. It would be easy to ask for a document and then destroy those servers that offered it.

That is why Eternity Service utilizes algorithm known as onion routing. This algorithm is designed so that the communicating entities do not know each other's identity. Using this algorithm the messages are sent through a number of hops (Mixes) in such a way that every Mix on the way knows just the address of its predecesstor an successtor and has no idea of what its position in the chain is. The only exception is the last Mix that recognizes that he is last and passes the message to its Six.

In the header of any message there is stored encrypted information about the address of next hop. Mix decrypts this info after receiving the message from network and finds out where to send the message next. The next Mix repeats this procedure and forwards the message further. Because the private key of a particular Mix is known only to this Mix, it is really hard for the attacker to find out he address of the next hop and even harder to find out the final destination of the message.

Forwarding message through many hops gives us quite a good protection against traffic analysis. It is also important to choose the right Mixes to communicate through. Their distribution should ensure that they are not under control of one authority, that could get an idea of what is going on by looking at the traffic routed between its Mixes.





Another problem that comes with increasing the length of path through Mixes can be disfunction of one or more Mixes on the way. This is more noticeable under ocasions when the path of Mixes is specified in advance and shoul be valid for a long time. To work around this problem we implemented a more sofisticated variant of onion routing. In our implementation there is duplicit information in each peel of message header that specifies multiple Mixes that the message can be sent to. The message can be in case of failure of one Mix routed through a different path. Choosing the next hop randomly from those specified in header can help to distribute the load between several Mixes. This has the effect of traffic analysis being even harder.

In our implementation we send the message to the first available Mix. Changing this behaviour to the random one means easy modification of Sender object. It would be nice to introduce a check that would prevent inserting a Mix certificate into a peel twice. This should be implemented in MesageCreator (Mix) in methods MakeMoreOnionLayers and similar. Onion routing is mostly implemented in object Translator that tightly cooperates with MessageCreator.

11.2. TCB

It has not to be technology or hardware what can be the weak place of the system, human factor plays its role as well. Such a weak place could be e.g. an administrator of an Eternity Server - server which maintains stored files. What will do an administrator who is threatened to delete certain file from his computer? With a knife on his throat he will delete it almost certainly. Such a possibility would lower the trustworthy of the system in a large way. But what will he do not given a chance to even find out what files are stored on his server? Surely, it is possible to physically liquidate the server, but there is no software capable of avoiding this, on the other hand not even government would probably dare to keep someone away from doing his

business by demaging his property.

The situation, when even an administrator has no chance to find out what data are stored on his server, can be reached by a simple cryptographical device, which will store its private key purely in hardware and which will be able to realize ciphering algorithms (including operation that will get data ciphered with device's public key and return the same data encrypted with some other key not letting the data to be seen decrypted from outside the device). This device we call TCB - Trusted Computing Base.

Using the mentioned mechanism it is possible to reach the state when even the administrator of a file storing server does not know what data it maintains. Data will come encrypted with public key of HW device, which will decrypt them using its secret key and encrypt them again with another key in one step, thus decrypted data will not leave TCB. After finishing this the data are ready to be stored on a disk. So our goal is reached. It is possible to use the same mechanism on any data which need to stay hidden to an administrator's eye.

TCB has to have a way to remember what key it used to store what data. These realtions must be kept outside the TCB's hardware, because TCB has not enough memory on its disposal. But to hide even this information we use the key stored in TCB's hardware as a master key of a key hierarchy, which is used to encrypt all information that cannot be seen by server administrator.

11.3. Used cryptography

Security of the whole Service is ensured by using good cryptography algorithms. We make use of ciphering in the Service for:

- Privacy of transmitted information because of the great importance of the contens. This is needed for ensuring anonymous communication.

- Processing messages during Mix traversal. It is needed for data to look different when it go out of Mix. This is defense against traffic analysis.

- Privacy of stored data (inside Eternity servers). This arrangement defends against outside enemy (enemy wants to destroy the data) and also defends an administrator, because he doesn't know what kind of data is stored inside his Eternity server. Administrator can be defended only if a special hardware is used (see TCB).

In our implementation we use combination of symetric and asymetric cryptography. Asymetric ciphers are used mostly for privacy of transmitted symetric keys, symetric ciphers are used for privacy of the data. This combination was choosed because of greater efficiency with almost the same level of security.

11.3.1. RSAEuro and RSARef

In prezentation included in distribution (ppt format) we are liing a bit. We use RSARef (not RSAEuro). RSARef is copyrighted by RSADSI, so we didn't want to show off that we had stolen it. RSARef was changed a bit (one or two function from RSAEuro toolkit was added, look for name ``Pechy'' in the code, some ``extern "C"'' was added too). RSAEuro wasn't used because of (probable) errors we encountered.

We use RSA with 512 bits, there is some macro with number 512. It may work with a change into number 1024 with no another hacking, but don't rely on it. From symetric ciphers, we use only simple DES. See object Cipherer for

more information.

11.6 Anonymous accounts

For state institutions (such as police, court of law, ...) could be an easy way to trace users or Esos' administrators by finding out owners of accounts geared to the Eternity Service. Solution of this problem consist in using anonymous bank accounts. Whole financial transactions in the Service are carried out only through such accounts and that is why it is impossible find out their author or recipient.

11.8. Defence against traffic analysis

Traffic analysis is an attack taken by mostly passive attacker, who is truing to guess what is going on and who is communicating with whom based on watching the traffic. The attacker could watch some special kind of messages. That is why all the traffic going through the service is encrypted - it is hard to recognize the type of any message.

The attacker who has control over a larger part of network would be able to trace a message from the source to its destination based on its never changing content. That is why the message is reencrypted at every hop and so the content changes randomly. As the content changes the only way to trace the message is based on the size of message.

To randomly change the size of message is technically infeasible. It is easy to enlarge a message but splitting message in the middle of the way is not possible without weakening the protocol.

We chose to send just uniform chunks as it is quite straightforward and effective. What the attacker sees are random fixed size chunks that are changing at every hop.

As the messages consist of two separate parts - header and data - the reqirement of fixed random size has to be obeyed in both cases. The optimal size of header and data with respect to effectivity is subject to further discussion.

11.8.1. Data Padding

Uniform size of data is done by chopping message into several fixed size chunks (see Chopper in ix section). At the end of the journey of the original message is reconstructed by joining the chunks together. When chopping the message Chopper adds some additional headers to make joining chunks possible.

When going through Mix tha data is encrypted using a symetric key and so the content of message changes completely. The result of encryption is a multiple of key length so if the input already was then the size does not change.

Chopping the message and addind headers brings a risk of revealing this info in the middle of path when sending to an onion (see section onion routing for further dicussion - "Middle attack").

11.8.2. Header Padding

When keeping fixed size headers we are in slightly more complicated situation.

When Mix decrypts the header it takes some information away and so the headers would decrease in size as the message flows through Mix chain. An attacker could so see that messages with short headers are near to their destination if not directly heading to it.

Fortuantely the size of info taken from header is known and so it is possible to append some rndom data at the end of header making it hte same size again. It is not possible to distinguish this padding from the rest of onion as it is encrypted and as this has character of random data. The last Mix recognizes he is last by seeing a special string in destination address and so the accuulated padding gets ignored.

11.8.3. Analysis of amount of traffic

It is hard to trace message based on their content or size but it is still possible to make some conclusions about roles the machines play in Eternity Service based on the quantity of messagesand its dependency on time.

11.8.4. Traffic padding

A system resistant to this kind of analysis would have to produce constant flow of data and would have to receive constant one too. In reality it is impossible to achieve this and it would be really wastefull to send so much padding messages to keep the flow constant even on lowly used systems.

It is necessary to find a resonable balance between security and effectivity. Mix has to operate in real time and so it is not possible to keep message for any amount of time and so the means to pad traffic are limited.

We can generate fake messages and delay messages. The question is whether it is not possible to squeeze th etraffic so that the gaps will get filled with messages that can not get through in time. The algorithm that adjusts the throughput of messeges is implemented in Padder object that has complete control over the data going through Mix.

Maybe it would be useful to consider the possibility of using dummy net to shape the traffic to some long term average.

The messages going to Six are not subject to traffic padding as this communication is expected to go through a controlled network or take place on local machine.

11.8.5. Summary

What is traffic analysis and how does Mix protect itself against it? The message going through chain of Mixes is changing at every hop but its size remains the same - so the messages are indistiguishable from each other.Every Mix knows just its predecesstor and successtor and has no idea of what its possition in the chain is. Together with padding messages and shaping the traffic padding presents a mean to fight against traffic analysis.

11.9. Time synchronization

Quite a serious problem for Eternity Service is time synchronization. We need a robust time synchronization protocol. As Eternity Service stores documents for a specified period and then deletes it there is a danger that an attacker

would try to persuade the server that the time to delete the document is here.

The basic condition is that we will never trust a third party and we will never accept any result comming from some consensus that was not initiated by us. It is possible for an attacker to initiate a time sychronization between a number of his machines and a server that he wants to persuade to delete files - as he has control over the machines the result of synchronization will probably be what he chooses. But on the other hand we can accept just time synchronization results from groups that we have chosen (probably randomly as it is the most secure method).

We can synchronize with a bank but there is always a possibility that the Bank will have bad time (maybe unintentionally).

The use of external hardware is not that easy and may not be feasible for everyone. This also does not solve the problem with fake signals.

Our conclusion is such that the machines sholud be synchronized some accurate time synchronization protocol (NTP) against several sources and for security reasons the servers should perform additional synchronization and in case of a bigger difference in times obtained by these two means it should not delete any files and warn administrator.

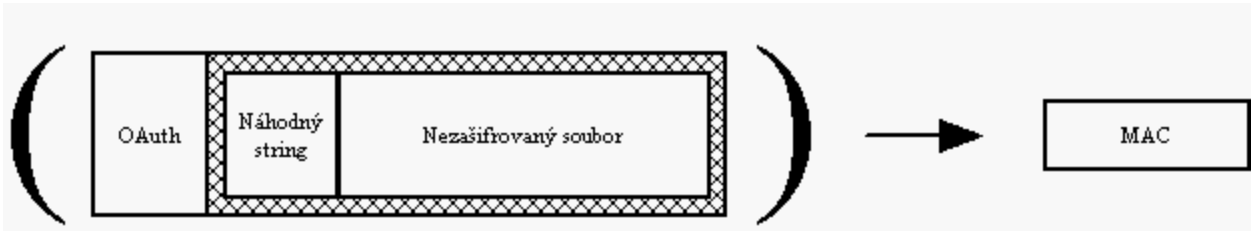
Some implementation of this idea shoul be found in Eso's objects TimeSynchronizer and Scheduler.

11.10 Verifing authorization of payments for data storage

This chapter describe a mechanism, which is provided by a server storaging data (Eso), an user accessed program (Client) and a server of financial institution (Bank). The mechanism was proposed to make impossible for Eso to pretend ownership of a file and for everyone else - mainly the user himself - to pretend being the Eso and get his money back.

The Eso does not get a file for storage in a plain text but in a following form:

The Client has put in front of its file a random string (which has been unique for each Eso) and the result has encrypted with its Secret Key. This way each Eso gets slightly different copy of the file. In addition for each payment in the sent payment plan the Client generates another random string (OAuth). Using this string he computes MAC (Message Authentication Code - an MD5 hash over the data). All pairs [OAuth, MAC] are sent to the Bank together with payment plan.



Also the Eso generates random string for each payment (SAuth). This string is secret to the Eso and the Bank and prevent to anyone else (especially the data

depositor) from masquerading the Eso.

Asking for a payment the Eso receives from the Bank challenge with OAuth to compute MAC. As an answer the server sends back the checksum (MAC) and the correspondent SAuth. The Bank compares MACs and SAuths with its values and if both pairs are identical, transfer money amount to the Eso's account.

Can anybody get a file in a plain text (i.e. not encrypted)?

Of course, yes. Together with the encrypted file from the first picture it is also sent a corresponding Public Key. With this key anybody can decrypt the file without any problems.

Can an Eso delete a file and keep only a small piece of information (for example the generated random string from the posted file) to be able to get money by pretending data ownership?

Definitely no. An Eso is prooving ownership of the file by computing MAC and an additional information for this it receives from a Bank not before date of the payment. MD5 garants that an Eso's administrator cannot store only a piece of given file or any other information (precomputed checksum, ...) to be able to prove the data ownership.

But the Eso could not pretend data ownership and in the date of payment it only ask for the file in the Service. After receiving the file from the Service (in a form described above - see the first picture):

- it can decrypt the file (by given User's Public Key) but is not able to encrypt it back with User's Private Key and so compute MAC.

- also the Eso cannot use the file as it is, because it does not know the secret SAuth.

So when the Eso deletes or modifies commited data it is loosing chance to get its payments.

Still there is a little problem: an Eso could store files onto an external data media and use them only in the "payment day". But the question is the motivation of keeping such devices out of the Service. On the top of it this "strategy" could become very disadvantageous by scheduling frequently payments in less amounts.

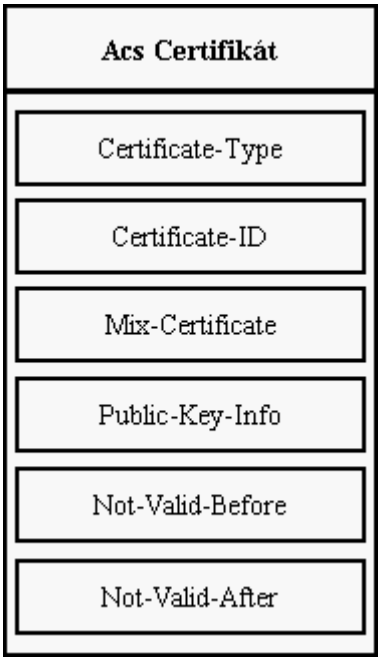
---

12. System realization

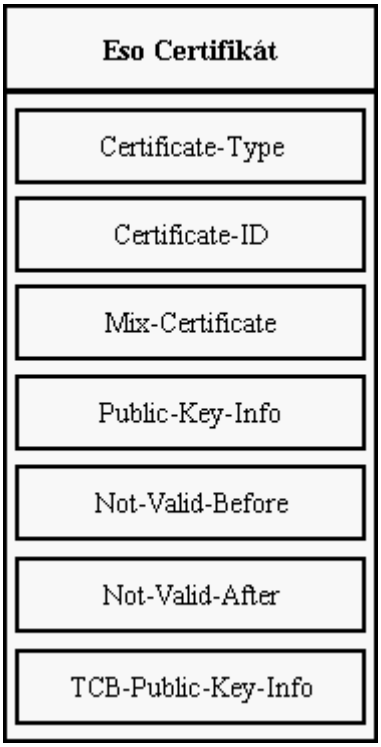
12.1. Identification of subjects in the Service

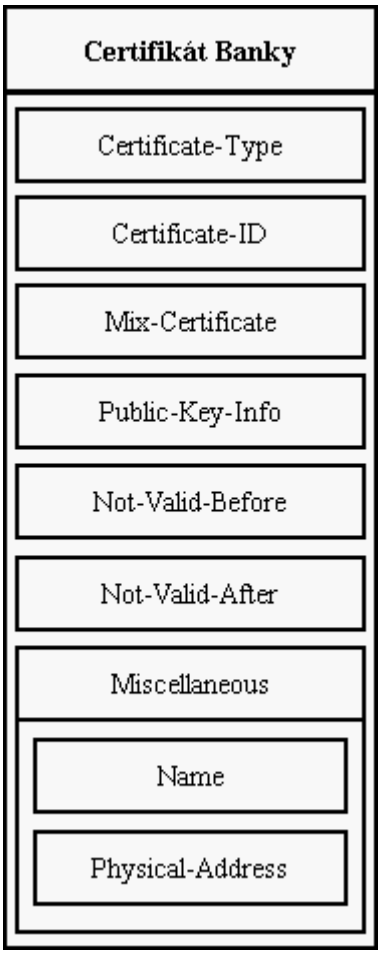
All servers running in the Service need a way how to contact them. We have to contact even Eternity servers that have to be maximally secret and hidden.

12.1.1. Certificates



Each server is identified by its certificate (acces certificate). This certificate contains for all servers compulsory item. This item is called Certificate-Type. The format of certificates for all servers is somewhere else.





12.1.2. Certificates for Mix and onions

In Mix's certificate there is IP address (or DNS) and port that uniquely identifie location of the server. IP address is recommended (with DNS is easier to locate the server). This advertisment of IP address is not

possible (generally) for Six that has to be hidden (ie. Eternity server). We solve this situation with so called onions.

12.1.3. Onion

We can compare an onion to generic address. Eternity server includes its onion to its certificate, Client sends (during searching for particular file) an onion, where to send answers to. Onion consists of peels where each of them identifies one hop on the way from sender to the owner of the certificate. Each of these peels specifies several recipients (currently 2) where is possible to send the message with the onion to. Each recipients denotes the next hop on the way.

12.1.4. Making the onion

If we hold an onion, we can always add some peels on the top of it. This way Client can secure itself against divulgement. If he wants to contact an Eternity server, he add some peels on the top of the particular Eternity server's certificate. Client use this defend on account of the situation where Eternity server is administrated with our enemy. The enemy gets a message that went throuhg some hops that enemy doesn't know, so it is a big problem for enemy to find out the sender. The onion for Six's certificates is always created by Mix, Six doesn't know the onion's structure.

12.1.5. Number of peels in the onion

Number of peels says how secure the communication channel is. On the other way, it is very important that all messages have an onion with the same lenght. This is due to the fact that enemy could choose messages that have short onion and could try to break them. When Mix adds some new peel on the top of a certificate, he should check whether the certificate given didn't get over the half of the length limit for onion size. Currently, Mix doesn't check it.

12.2. System components

12.2.1. Mix

12.2.1.1. Functions of Mix

Mix is a part of Service that is to provide services for anonymous communication between Sixes in Service. Mix is a ,,remailer clone'', resending messages of electronic mail. As opposed to remailer, Mix is working is real time and therefore we can't use all techniques that can be used in classical remailers.

We want Mix to be able to route messages to virtual addresses that are reprezented by access certificates. We want Mix also to be able to generate these certificates and to publish them onto Acs servers.

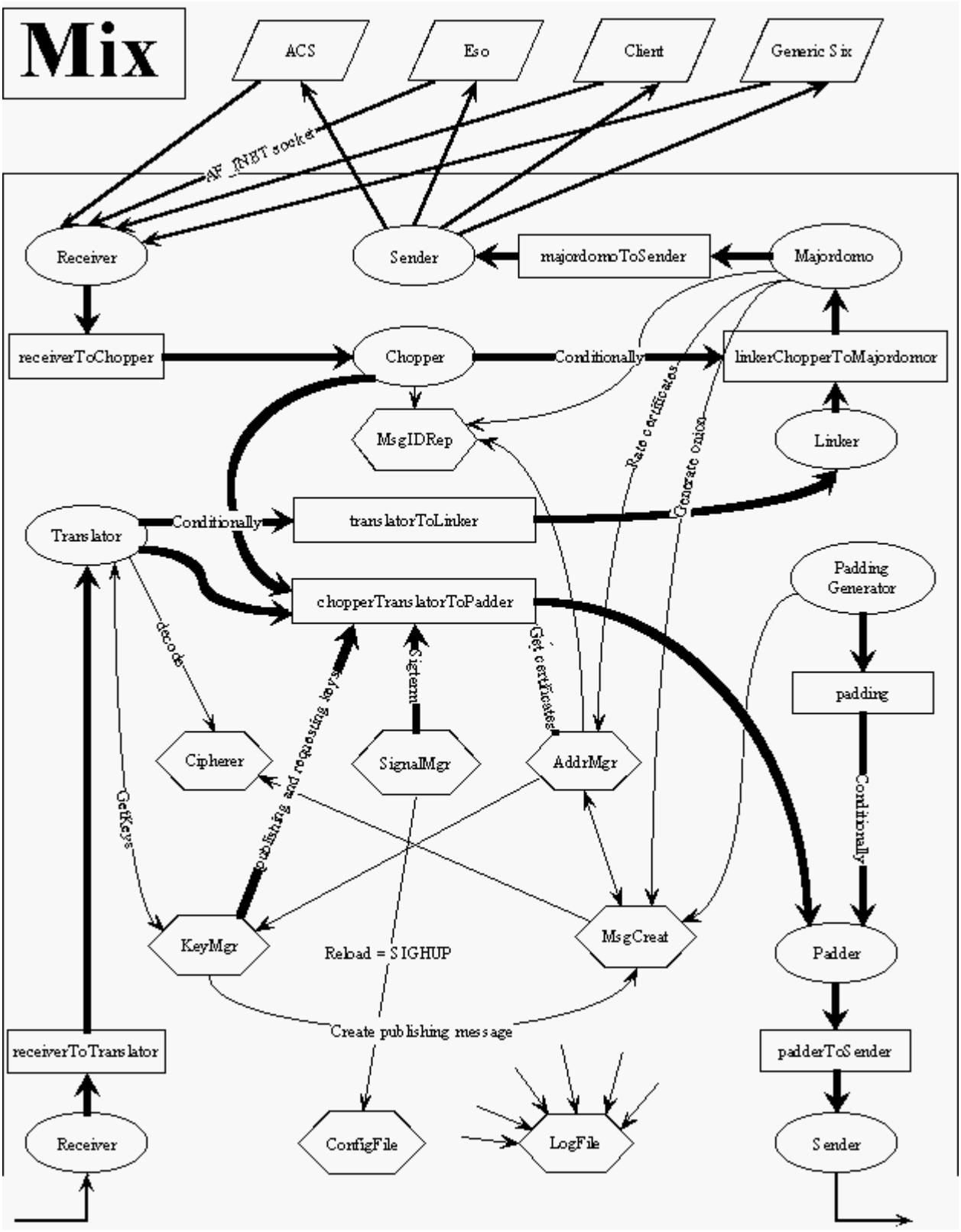
Mix has to provide anonymous communication. Ie., sender and reciever can't be easily traced as the message goes through the chain of Mixes. Therefore,

we have to implement a protocol that can withstand traffic analysis, delaying of messages, masquerading for someone else etc. The protocol in Mix tries to prevent from efficient traffic analysis. Other security mechanisms are included in protocol of communication between Sixes.

Mix is the low level layer of the Service, the layer that tries to provide anonymity for communication between opponents.

12.2.1.2. Architecture of Mix

Mix consists of several independent objects. Each of them provides exactly one function of Mix. Mix's architecture is shown in the picture below.



Mix consists of static and dynamic objects. Dynami objects have a special method, called ::Run(). This method is executed in a separate thread. Static objects have no such method, just provides some services to another objects. Messages processed by Mix are moving along several ways inside of Mix. These ways create 2 main loops. Message that is just forwarded by Mix is receiving from the Net by Receiver. Translator then gets it and finds out that the message is going to be sent to the next Mix. Translator then puts it into the queue leading to Padder. Padder resends it to Sender. Sender is the last objects in this way, it sends the message to the Net. This situation is just due to making enemy confused where the message is. The second situation is one

that Mix is the very receiver of the message. In this case, the message is received by Receiver, processed by Translator and resent to Linker. Linker waits for all chunks that makes the message and than the whole message is forwarded to Majordomo. Majordomo decides what to do with the message next. If the message is for a Six above this Mix, message is resent to Sender. When Six wants to send a message, he sends the whole message to "upper" Receiver in Mix. Chopper chops the whole (unlimited in length) message into fixed sized chunks (Chopper uses MessageCreator for creating fixed messages from chunks) and sends the cooked messages to Padder. Padder mixes these messages with fake messages (traffic padding) and forwards messages to Sender. Messages that are to be processed by Mix (eg. certificates creation) are managed byMajordomo.

12.2.1.3. Implementation of Mix

12.2.1.3.1. Mix

Object Mix is the main object of the whole Mix server. This objects have several objects that implements Mix's functionality. Mix as an object just creates these objects and runs Run methods in separate threads.

12.2.1.3.2. Translator

Translator together with MessageCreator object implements the core of onion routing algorithm (for more information on this topic see Onion routing). Translator receives messages from Receiver object and removes the top peel. The peel removing is accomplished this way: first Translator decodes symetric key (with Mix's private key) found in Recipient field in the message and this symetric key is then used for decrypting the rest of the message header. In header, there is another symetric key that is used for decrypting the message body. Now Translator has enough information about where the message is about to go next.

Because of the fact that it is needed to decrypt the symetric key as a first action and the that is used to decrypt it is Mix's private RSA key, nobody can get information about the next hop but Mix.

From information decrypted from the message body Translator can decide whether to route the message to the next hop or whether the message is for some Six above this local Mix.

If the message is about to go to the next hop, before the message routing it is needed to encrypt the message body with a symetric key that was found in the decrypted header. The body now doesn't look like the body that came and it is much harder to trace it (header is also changed).

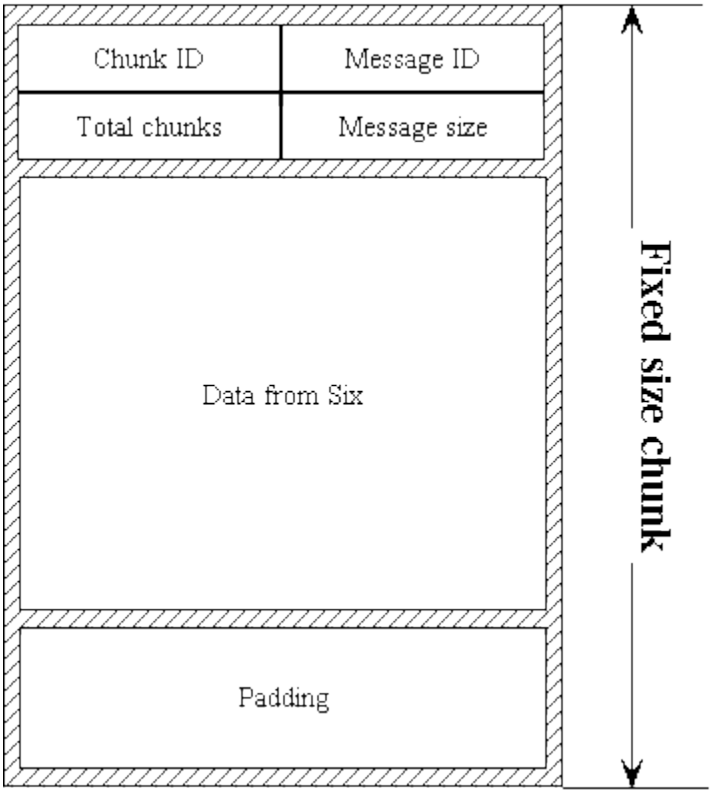
Translator is one of the objects that implement important functions of Mix as a communication server, ie. making traffic analysis much harder. Decrypting of the message is fullfiled with help of KeyManager and Cipherer.

12.2.1.3.3. Chopper

Processes all the messages from its input queue (from Receiver). If the message is destined for the local Mix, it is most probably some kind of request from Six and so the message is passed to Majordomo that will takecare of it. If it is a request for sending data ,then chopper chops the message into uniform chunks - padding the last one if necessary. These chunks are then concatenated with headers that contain info about total number of chunks,

chunk number, message length and message ID. These headers are necessary for reconstructing the original message at destination address.

MessageCreator then prepends onion before chunk and adds some additional peels. Now is preparation of chunks finished and all the data can be passed to Padder.



So that the additional headers are not visible in the middle of path they are end to end encrypted using an additional asymetric key - see MessageCreator.

Optimal chunk size is not known as well as header length.

12.2.1.3.4. Linker

Processes messages that it gets from Translator and inserts them into its internal structures. These messages are in fact blocks made by chopper. Linker uses the additional headers in chunks to collect all chunks belonging to one message and reconstruct the original message. The final message is then passed to Majordomo.

Linker should implement some kind of timeout to discard messages that are not complete and some chunks probably got lost. Else it would eventually fill with incomplete message.

12.2.1.3.5. Padder

Together with PaddingGenerator they implement a strategy for traffic padding and traffic shaping. We tried to implement some strategy into these objects but it did not work so it is not used by now. Maybe it would be a good idea to try to use a tool like dummy net to shape the traffic to some average - the peaks being cut would then fill the gaps.

12.2.1.3.6. KeyManager

Takes care of keys of the local Mix. Generates new keys. Expiration of keys is not implemented yet. Cooperates with AddressManager.

12.2.1.3.7. MessageCreator

MessageCreator is the object that can build whole messages or onions only. AddressManager makes use of his service for creation of messages used for certificate publications, Padder cooperates with MessageCreator in creating padding traffic, Chopper asks him for creating of messages from so called "chunks", that Chopper chops from the data of any length (that data that is about to send by Six).

12.2.1.4. Onion creation

Each peel of an onion has to be able to open only with decrypting of a symetric key that was inserted in the peel above the current one. For decryption one has to use Mix's private asymetric key that is in turn in the chain of Mixes creating the secure channel. Ie. that during the creation of the onion we have to know all the certificates for Mixes that will be on the way. All peels are during the addition of the next one decrypted. This way, Mix anywhere in the chain can find out just the next Mix on the way. This solution to traffic analysis problem is called Onion routing. See Figure somewhere else (###).

Before we begin to build an onion, to the inner peel we have to save all symetric keys that will be used for building the onion. In this bottom peel there is also saved a freshly generated private key. The public key form the same pair is put on the top of built onion. See next para for more information

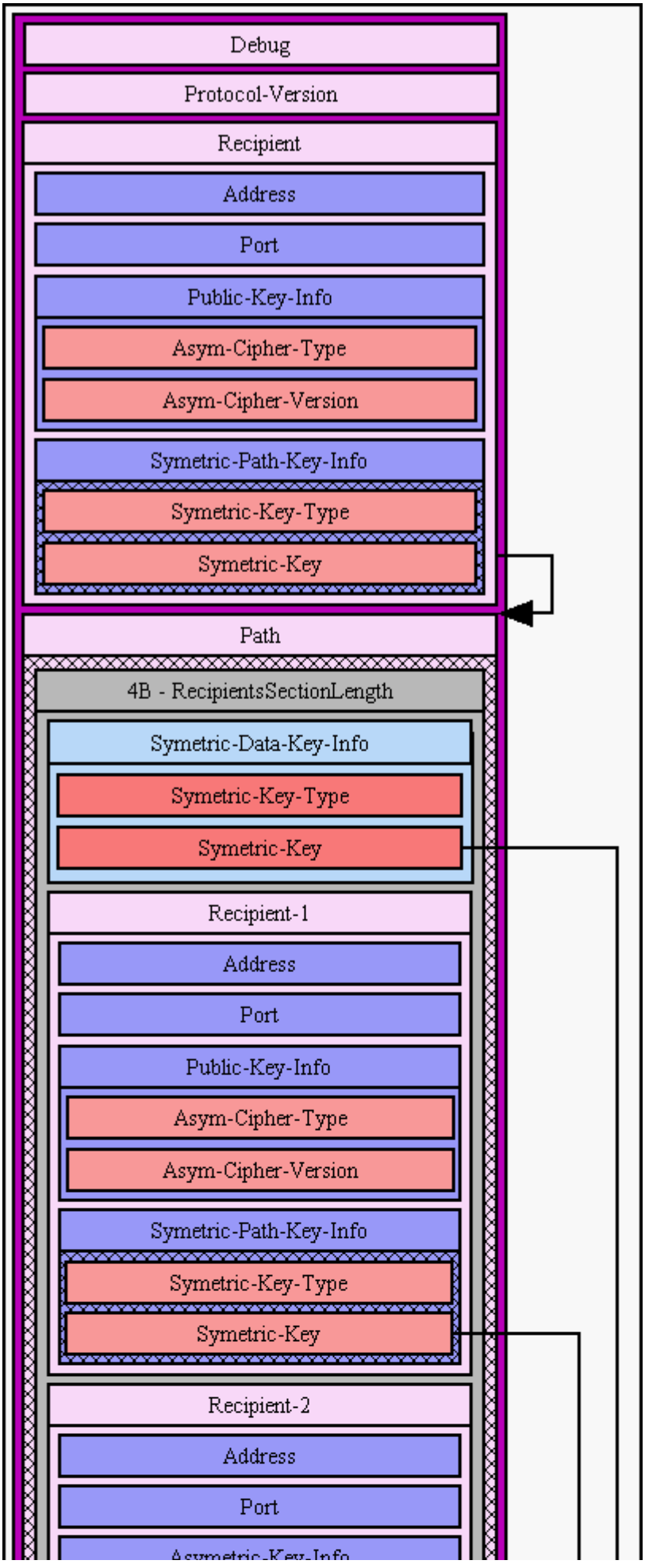
---

12.2.1.5. Message creation

Creation of messages has two parts:

- during the first phase Mix has to add some peels on the top of an onion that is given by Six. Adding some peels is not neccesary, of course. But it is needed if Six wants to feel secure. For more info, see Identification of objects. Public key that is on the top of the onion we save aside, Mix will use it for processing of given data.

- the second phase is processing of data that is given by Six to be sent to the subject identified by the onion. As it was already said somewhere, body of the message is encrypted in each Mix that is on the way. The reason is that we need the message to look like different message as it goes out of the Mix. See next para for more info.


Debug
Protocol-Version
Recipient
Address
Port
Public-Key-Info
Asym-Cipher-Type
Asym-Cipher-Version
Symetric-Path-Key-Info
Symetric-Key-Type
Symetric-Key
Path
4B - RecipientsSectionLength
Symetric-Data-Key-Info
Symetric-Key-Type
Symetric-Key
Recipient-1
Address
Port
Public-Key-Info
Asym-Cipher-Type
Asym-Cipher-Version
Symetric-Path-Key-Info
Symetric-Key-Type
Symetric-Key
Recipient-2
Address
Port

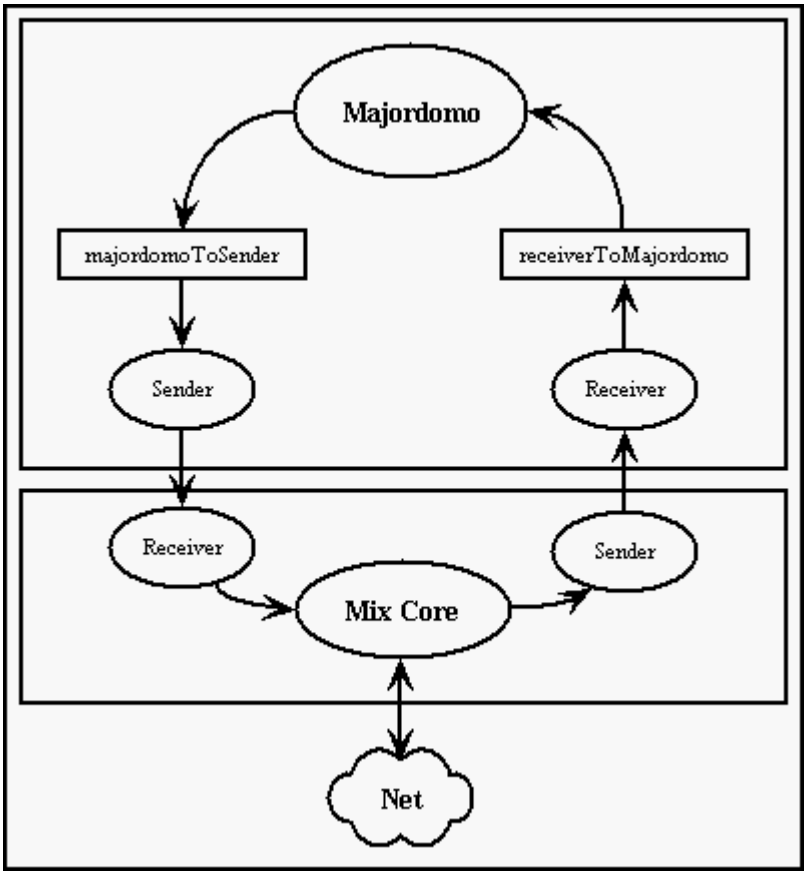
12.2.1.6. Data processing

When a Six sends data using an onion as a receiver address, Six can ask Mix for adding some layers on the top of the onion. Mix generates the same number of symetric keys as is the number of peels that are going to be added. The data part of a chunk is decrypted with all there keys. These keys are then included in the chunk's header (each key in one peel). When the chunk (fixed message) goes through the chain of Mixes, these keys will be used for encrypting the data part. Algorithms for symetric encoding implie that it wouldn't be safe to decrypt the data first (ie. to multiply decrypt plain data). These data needn't be random and from decrypted data that wasn't random we can get the plain data easily. From the information given here, in the middle of the chain of Mixes the data are in the plain form (ie. the form that was given by Six). During the second part of the chain, the data part is still encrypted in each Mix, so when the chunk comes to the last Mix, the data part is multiply encrypted with symetric keys that was previously inserted into the lowest layer of the onion (see Creation of onions). This implies that is is easy for the last Mix to get the plain data (after using the private asymetric key that was also found in the bottom layer). For further information, see section describing Translator object.

MessageCreator is the main objects that is involved in algorithm of onion routing. He creates messages in the way that makes enemy confused about the real movement of messages.

12.2.2. Six

Into the design of the Eternity Service we incorporated the possibility to connect any server, which is communicating according to a certain protocol, to the net of safely communicating Mixes. A core of such a server was implemented and is called Six. Eso, Bank and Client are currently implemented using this core, which encapsulates creation of basic threads of a server and methods that serve messages coming from underlying Mix.



Six includes these threads:
- Receiver
- Majordomo
- Sender

Receiver gets messages from Mix through a socket, which means that Six and underlying Mix can be run on different computers. Majordomo pareses and serves incoming messages and Sender is responsible for sending messages generated by Majordomo through a socket to Mix. Queues are used to transfer messages among Receiver, Majordomo and Sender.

The sceleton of Majordomo in Six takes care about this communication basics with Mix:
- asks Mix to generate an onion and calls an Six's abstract method to generate new access certificate
- asks Mix for access certificates of other Sixes and calls an Six's abstract method to take care of them somehow
- from Mix it receives a data message sent by another Six
- for Mix it prepares a message to be sent to another Six

Communication among Sixes is always secret because of our safety requirements. Every single message is encrypted with a public key which is a part of the recepient's access certificate. Therefor it is needed to encrypt all messages of Six->Six type which are going to be distibuted into the Service this way. Six Messages received by another Six must be than decrypted by appropriate secret key which makes a key pair with public key from access certificate the message was sent to. Both these operations are supported in SixMajordomo. The incoming messages decrypting method is defined as abstract due to differences in secret key storage algorithms in different Sixes.

As it was mentioned above, one of the key features Mix does for Six is that it realizes all communication with the whole Eternity Service. An Eso administrator does not take care about the way messages are sent by Mix, and on the other hand Mix does not care about the contents of messages it

is asked to send. The SixMajordomo's task is to prepare a command for Mix that consists of data to be sent and of an onion (which should be understood as a generic address) data should be sent to.

Six is easily configurable by its configuration file, which name is given to Six in its constructor. Configuratin file is described in section ###SpolecneObjekty. There are data necessary for a Six to run in the configuration file, such as:
- name of a Six,
- IP address and incoming and outgoing port of an underlying Mix,
- path to a directory where Six's specific persistent data should be stored.

The name of a configuration file is given to Six as a parameter on a command line.

To simplify interaction among objects that are parts of Six's descendants we implemented their ancestor - SixOffspring, which only stores a pointer to its parental Six. Thanks to this pointer all these objects can call each other's public methods.

Six is easily configurable core of all executive servers, is capable of receiving and sending messages from and to Mix and thus from and to surrounding Service. To implement new executive servers that could transparently use the principals of anonymous communication through a net of Mixes means to override and/or add characteristic messages serving methods in Majordomo.

12.2.2.1. Eso

12.2.2.1.1. Functions of Eso

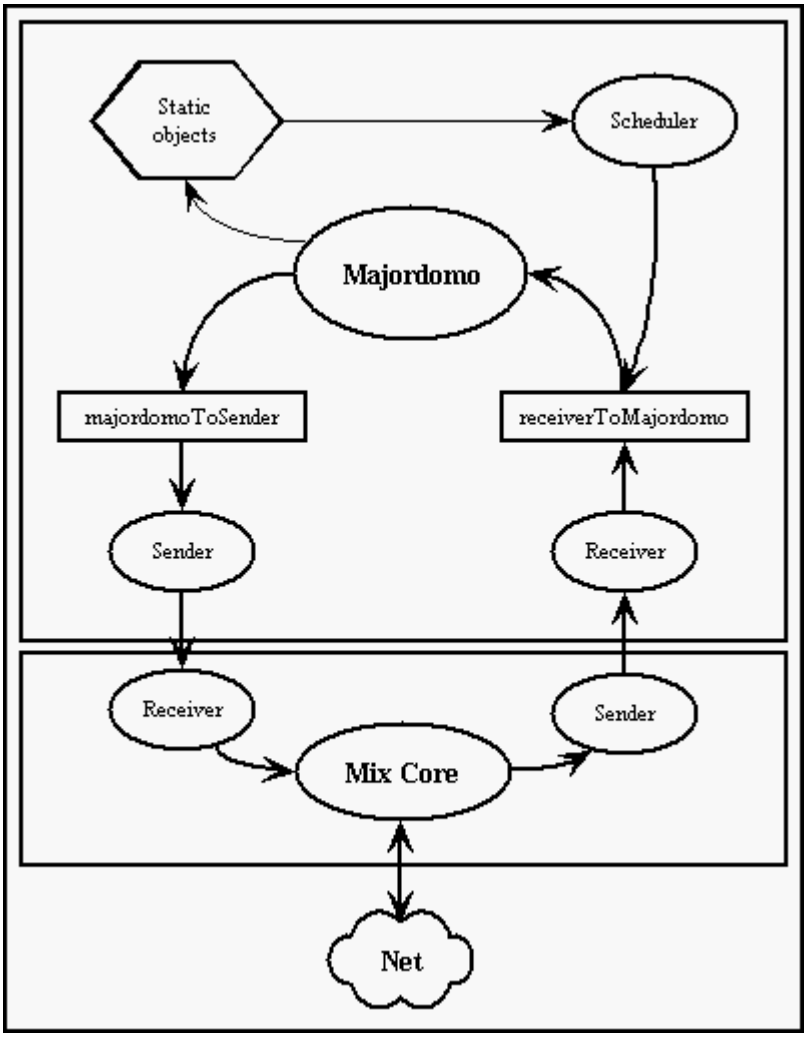
The main function of an Eso is safe saving of files. It should be done in a way that even the administrator of the server doesn't know the contents of files saved on the Eso.

Along with this feature the files must be accessible for the users of Eternity service. The Eso must be able to search through the database of files and send back requested data. Therefore each file is saved together with keywords that are used in requests for search in the Service.

As a motivation for the administrators to run Eternity servers a mechanism of paying for stored fiiles is proposed. Eso must be able to send a request for pay for stored files. Again this must be done in a way that the money is sent only to a server who really stores a file (see Protocols section).

Because all files are stored on Eso only for a certain time, the time synchronization mechanism must be implemented. Without this feature the easiest way to force Eso to delete a document would be to move its clock.

12.2.2.1.2. Architecture of Eso



Eso has a role of Six in the Service. The basic function of Six are extended with functions that serve the messages in the Eso-Client, Eso-Eso and Eso-Bank communication. The thread for generating parametrized timeouts is added. This thread generates timeouts, that are ordered by other objects while serving messages.

All operations done over the incoming messages are quite simple and most of the time is spent on ciphering that is realized by a synchronnous device (TCB). That's why even our realization of message serving is done in a synchronnous way.

12.2.2.1.3. Implementation of Eso

Eso uses the basic functions inherited from Six. The Mix-Six incoming commands are served "automatically" - in Six. The messages from other Sixes are served in methods of Eso Majordomo using the static objects of Six. The timeout messages from Scheduler, that are also put to the input message queue of Six, are served in Majordomo too. The structure of the messages comming from Service to each Six is compulsory and is described in Protocols section. Eso doesn't serve messages that don't have this structure (ignores them).

Description of the classes used in implementatiion of Eso follows.

12.2.2.1.4. Objects of type ACManager

Eso includes several objects of type ACManager. Each of them controls a table of access certificates of one type. The main functions of ACManagers are
- inserting new certificates into the table
- returning requested number of certificates
- deletes expired certificates (each certificate has an expiration date)
- saves information about the optimal (minimum and maximum) number of certificates
- returns information about how many certificates are missing to optimal state
- saves information about how often to test the number of non-expired certificates

None of the ACManager generates the certificates of the Eso. These are generated in Majordomo methods.

12.2.2.1.3.2. Allocator

Allocator is used every time a request for storage arrives to Eso. It decides whether Eso will agree or disagree with file storage. Current implemetation of Allocator simply tries to allocate requested space on a disk and if succesfull it agrees with the request otherwise it refuses to store the file.

All information about allocations processed is kept in a table, which has ID of a transaction, in which allocation was requested, as a primary key.

If Eso does not receive the file within certain time period, then Allocator is called to free allocated space. If Eso receives the file it is waiting for, Allocator is asked whether there is an valid allocation for file it received.

12.2.2.1.3.3 Banker

This class manages all activities geared with payments for data storage:
- generates payment plans
- manages tables of payments and "currently carried out" payments

While the Eso store the file the Banker registers important information about payments. These will be used in data ownership prooving in the date of the payment.

The Banker communicates directly with the Scheduler. It is setting timeouts, which will inform the Banker that it is time for asking the Bank for money. Than it provides communication with the Bank.

12.2.2.1.3.4. Eso

This class encapsulates the main functionality of Eso. As a Six descendant it runs all necessary threads and creates all static objects as Allocator, Banker, Finder, etc. Because it is derived from Six, it is assigned a configuration file with specific information. This file can be reached from all objects that are owned by Eso (such as Allocator, Banker, ...) and that where derived from SixOffspring (see ###Six).

12.2.2.1.3.5. Finder

Finder is used for searchhing and downloading files.
It is a very simple object encapsulsating two tables:

- forward table (where to send file found on other Eso in a tree search)
- reply table (what ffids were already sent - used for cutting the serch tree through)

The tree search algorithm is described in Protocols section.

Note:
Finder as well as the methods in Majordomo using finder are completely implemented, but not fully tested (we didn't test the search tree).

12.2.2.1.3.6. Scheduler

Task of scheduler is setting of timeouts and sending messages to the queue between Receiver and Majordom when timeout is expired. Scheduler can send some information with expired timeout.

Expiration time is possible to set absolutely (accurate time) or relatively (time from actual time to expiration).

By this mechanism is implemented starting of events which is periodical (for example time sychronization) or ending of operations that have time limit or that can be timeless.

Important feature of timeout is possibility to add information which is send to the queue between Receiver and Majordom. Parameters of timeout's setting:
- time of expiration - absolute or relative
- command - identification of timeout's type
- data - added information

Setting of timeout is implemented by inserting the record into the sorted table. This table is sorted by expiration of timeout and timeout which expired first is also first in the table. Searching expired timeouts is realized by searching table in timeless cycle. This cycle runs in stand-alone thread and always is gone to sleep for some time because no operation is time critical.

12.2.2.1.3.7. TCBWrapper, SWTCBWrapper

TCBWrapper is an abstract class telling what methods must provide a concrete TCB implementation. In our implementation of Eso there is only SW emulation of TCB device so we needed to implement a descendant of TCBWrapper - SWTCBWrapper. SWTCBWrapper maintains a table of stored files and a table of certificate keys. Both tables are kept encrypted with symetric keys. These table-keys are part of hierarchical key management and they can be obtained only having the master key of the hierarchy.

All this information must not be reachable outside TCB, which is (thanks to hierarchical key management) reduced to simple request that the master key must not be reachable outside TCB. But when there is no TCB device in our implementation of Eso, we keep the master key as a private attribute in SWTCBWrapper. But because there is no special hardware to store the master key in, we simply store it on a disk, which is not sufficient.

TCBWrapper must provide these services:
- store a file
- find files and return their headers (describing what files were found)
- get file (decrypting it with TCB private key and encrypting it with Client's public key all that in one step)
- generate access certificate keys
- decrypt with access certificate's private key

See ###tcb_diskuze how we designed what and how HWTCBWrapper could work.

12.2.2.1.3.8. TimeSynchronizer

This Eso's object compares system time and time of others Esos periodically. The problem of found out an accurate time is written in detail in section

---

12.2.2.1.3.9. Transaction Manager

Though there is a lot of records in Eso, that are dependent one upon the other, we identified a need to keep all those records consistent. There are data valid only for certain time in Eso, which need a message from Scheduler telling that they are expired. If Eso is not correctly exited, it could happen that e.g. data are stored but there would be no timeout set for their expiration, so these data will never be released.

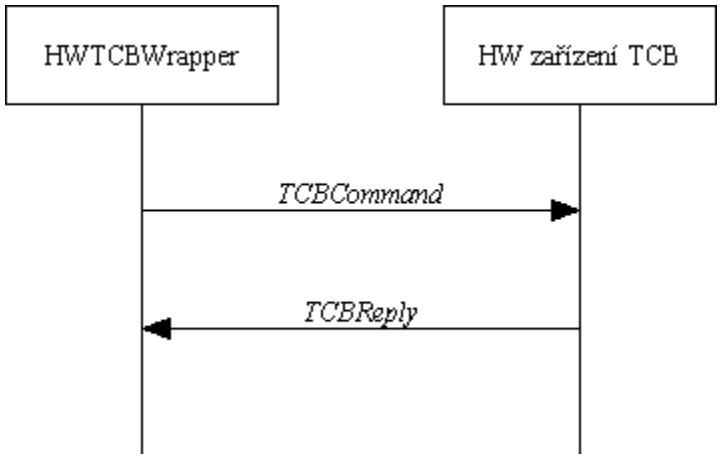
Both problems can be solved by TransactionManager. TransactionManager writes transaction ID to its log on every start of a transaction (which means the beginning of handling received message) and on every end as well. So when Eso is started it is sufficient to check for not ended transactions which are then rolled back. It means TransactionManager calls a Rollback method of all classes that has registered themselves as Rollbackcable (with a method of TransactionManager).

Current implementation does not parse the transaction log to find out what transaction have not been ended properly.

12.2.2.1.4. Discussion, alternative solutions

12.2.2.1.4.1. TCB

We made a proposal of HWTCBWrapper communication with HW TCB device, which is able to work with partial results:



HWTCBWrapper sends to TCB messages of TCBCommand type
TCB returns replies of TCBReply type

TCBCommand is of following structure:

- Command - information for TCB what it is asked for or what can be found in Data field
- Data

Field Command can be one of:

- Initialize - resets HW device
- predefined commands such as SaveFile, DeleteFileByPID - runs a TCB algorithm which leads to TCBReplies, in which TCB tells what it needs to get to succesfully finish the required task
- Data - indicates that field Data cares requested data

TCBReply is of following structure:
- Returning - indicates what comes in Data field
- Data - contents described by Returning
- Requesting - data TCB wants to get in the next step

If we want to work with partial results it is needed to distinguish between IDs that stand for standard (predefined) objects (such as Table of stored files, Table of keys, etc.) and IDs that stand for partial results. This takes part in Returning and Requesting fields. To solve this problem it would be sufficient to divide ID space into two parts: first (e.g. 0-99) would be reserved for standard objects, second (e.g. >99) for partial results.

12.2.2.2. Acs

12.2.2.2.1. Function of Acs

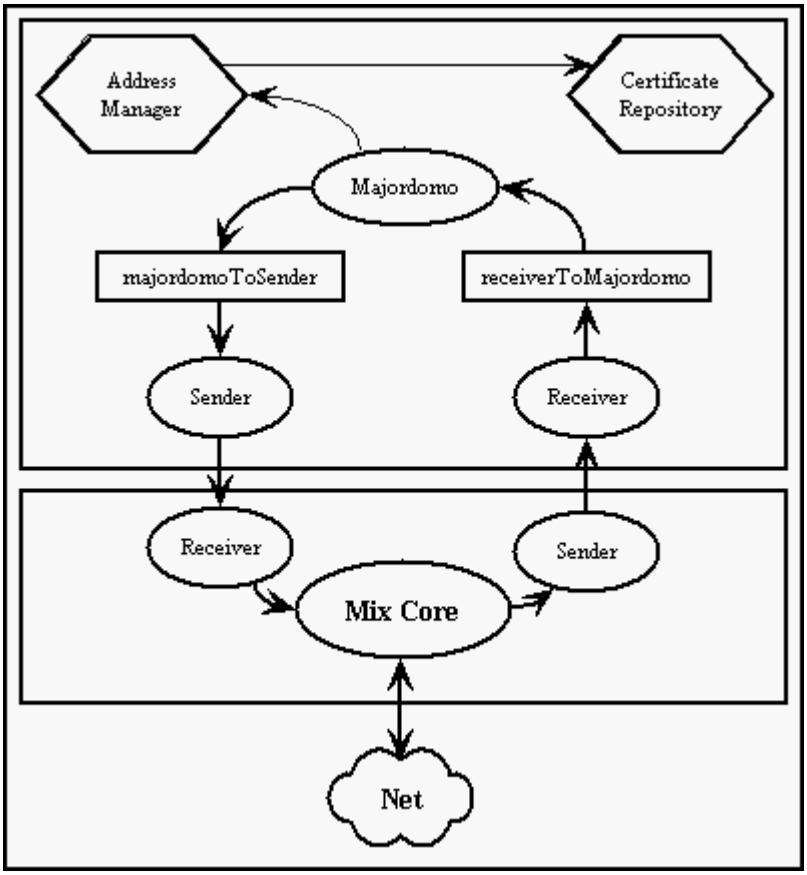
Acs is another Six that is running above the Mix (as each Six does). His function is certificate management. Aside from access certificates of Eternity servers (Esos), Acs has manage a database of Mix's, Bank's and and other Acs's certificates. Acs can manage certificates other Sixes (that are not implemented yet).

We want Acs to be able just serve request for returning some certificates (type + number is needed in the request) and to serve requests for storing certificates of Sixs that want to advertise themselves. Acs is the server that is a gate to Service for all other subjects in Service.

The very first idea was that we make use of an existing WWW server (Apache) and that Acs would be just a cgi script, that would cooperate with the WWW server. Because of format of communication and overall architecture of Service, we finally decided to implement Acs as a standalone Six.

12.2.2.2.2. Architecture of Acs

Architecture of Acs can be explained with the following Figure (###).



Acs consists of 5 independent parts:

- Sender, generic object that is used in all servers of the Service, Sender sends messages through the local Mix.
- Receiver, generic object
- Majordomo, this object is a core of Acs. Serves requests from Acs clients, ie. accepts and stores certificates or returns requested certificates (stored in Acs database)
- AddressManager, manages certificates in the database. AM is able to store certificates, search for them and return them.
- CertificateReposity, is called directly by AddressManager, is responsible for physical storage and physical searching for certificates.

12.2.2.2.3. Implementation of Acs

Sender and Receiver are objects that are the same for all Sixes, their implementation is explained in the respective section.

Majordomo is a object with a method Majordomo::Run() that is the core of Majordomo. This method contains neverending loop that is sleeping on the semaphore telling the number of messages from Receiver. Method servers requests CMD_PUBLISH_CERTIFICATE and CMD_GET_CERTIFICATES. The formats of respective messages are in programmer documentation.

CertificateRepository and AddressManager are static objects that are in this form used already in Mix. AddressManager is laid on the top of CertificateRepository, that is just a lowlevel tool for physical management of certificates.

At present, Majordomo doesn't check certificates at the moment when receiving them neither during the time when certificates are stored on the disc.

12.2.2.2.4. Discussion, alternative solutions

We decided to implement Acs as a standalone server that communicates with other servers using a defined protocol. It would be useful to make some kind of web interface to Acs for administrators that are setting up new servers - for secure operation of Mixes they need some certificates to start with. Including these certificates in distribution is not a good idea as the attackers could focus on the servers included in distribution and the load of these servers would be really high. Administrator could download some random certificates from Acs using the web interface - maybe from some Acses and then combine these packages to make sure he is safe.

By now setting up a new server is not that easy and involves quite a lot of manual and intelectual work.

12.2.2.2.4.1. Possible attacks against Acs and ways to defend

Acs does not check any certificates by now and as such can be subject to some attacks. This should be changed before Acs can be more widely used.

Enemy can generate access certificates with invalid addresses and fill so the Acs and cause a denial of service or fill Acs with certificates of his own controled Mixes and gain control over majority of traffic. The trust in service can be damaged badly by propagating invalid information.

Acs can protocet itself using thee techniques:

Check Mix certificates - it can try to send some message using the certificate and see if the Mix is really functioning. Such message would have to look the same as all other messages and go from Acs through tested Mix back to Acs. If it goes through the Mix is OK. If additional Mixes are used then the tested Mix has no chance to find out if it is a test message or a message passed through service. Such checks should be performed periodically.

Payment for publishing certificates - this would prevent anybody from flooding Acs with his own Mixes. This may be useful for certificates, that's functionality is not possible to check. But paying for publishing Mix certificates seems to be a little problematic.

Expiration time - Acs can have a maximum period of time for which it is willing to store certificate regardless of its expiration time.

12.2.2.2.4.2. Certificates maintanace

In case of massive and widespread usage of Service, it would be good to thing about making usage of an existing database engine. This engine would physically manage all certificates.

12.2.2.3. Client

Client is a module, that enables the user to upload, download and search for files including the communication with Bank. Because the Client communicates in the Service using a local Mix, we implemented it as a descendant of Six. Client is run always only to execute one operation.

12.2.2.3.1. Functions of Client

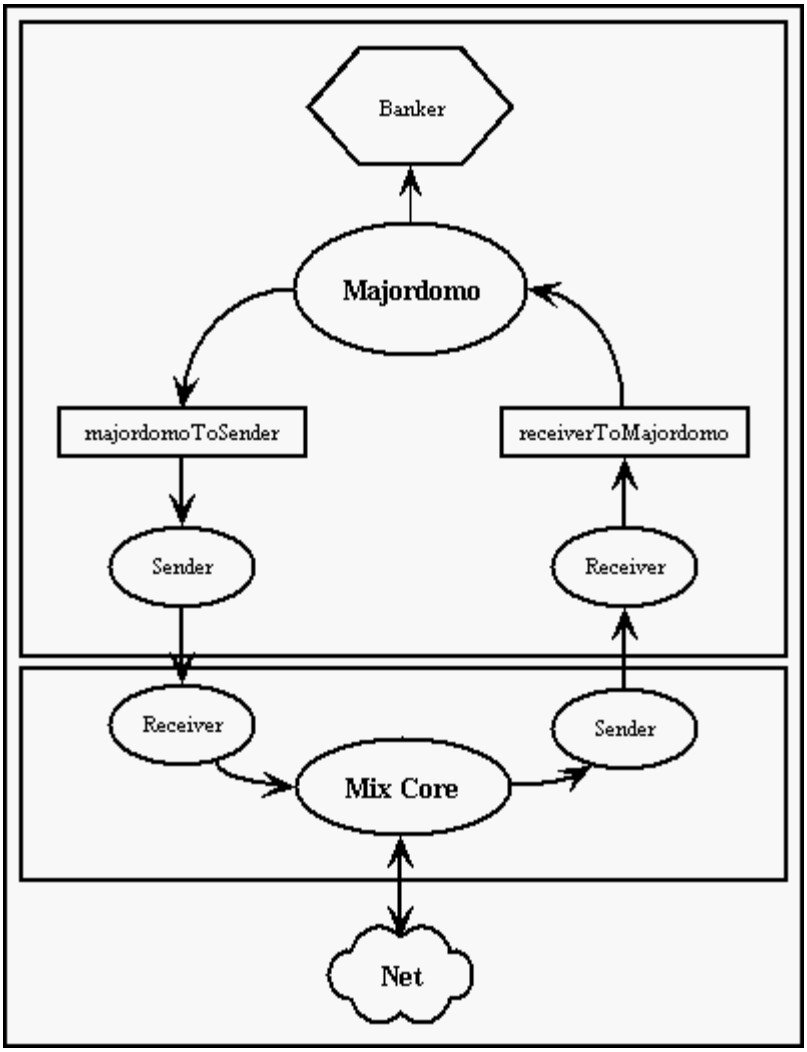
Client enables to perform these operations:
- upload of the file on a given number of Esos
- search for file headers according to the mask including keywords of the file
- download of the file according to the unique identificator of the file

Upload of the file has two files. First asks Client Esos for saving file. Positive responses to these requests are payplans. Client saves them to disk. Then Client sends file to Esos that agreed. (see Preparing the file for upload) The file is sent together with keywords and unique identificator.

Search and download are simple functions: Client sends requests to given number of Esos and waits for annswers.

12.2.2.3.2. Architecture of Client

Client inherits the basic objects of Six (Receiver, Majordomo, Sender) and adds one more (Banker). Majordomo's method Run is overriden (it is run only as a batch).



12.2.2.3.3. Implementation of Client

12.2.2.3.3.1. Majordomo

Object using SixMajordomo with added methods for upload, download and search, overriden method Run and many test methods (now mostly not functional).

12.2.2.3.3.2 Banker

When storing file this object manages payment plans and prepares information for the Bank.

12.2.2.3.3.3. Summary

This is "minimal functional version of Client". Three basic functions (upload, download, search) are implemented. It is run from the command line. The functions for communication with Bank are neither implemented nor tested.

12.2.2.4 Bank

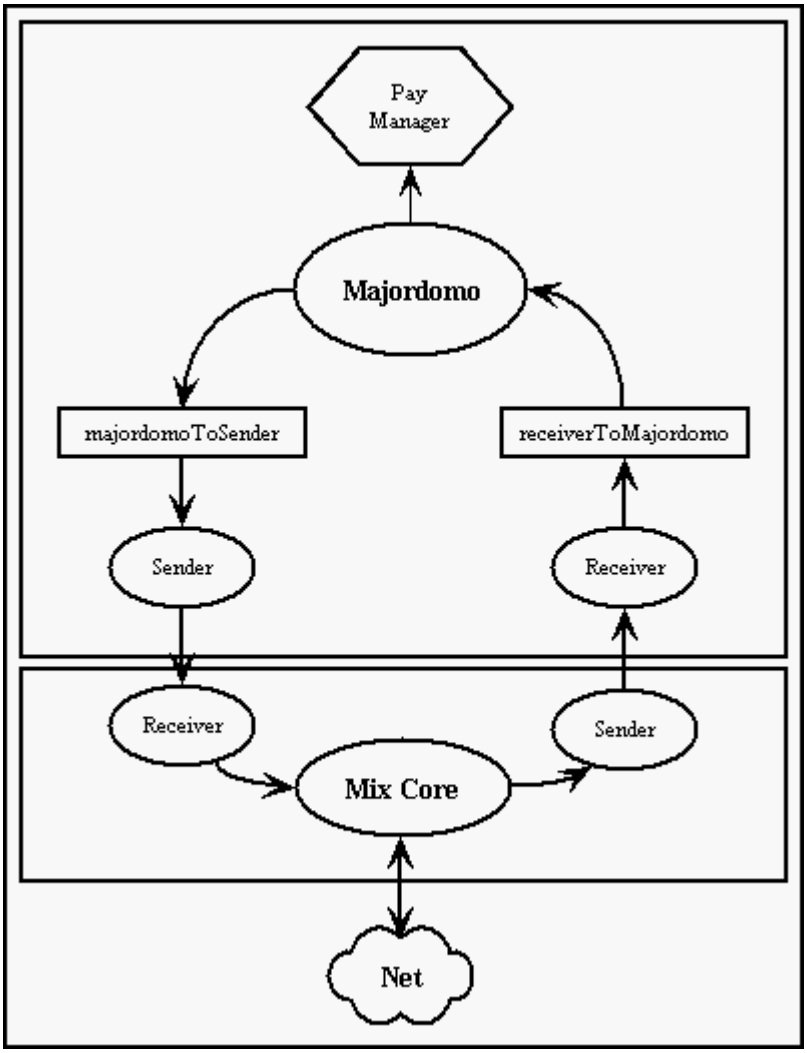
12.2.2.4.1 Functions of Bank

A Bank is a program managing payments and transactions in a financial institution geared to the Eternity Service. We assume, that this program will be connected to a system in real bank and with the assistance of its functions will be performed following operations:
- accept money from clients
- create anonymous accounts for them
- transfer correspondent money amount to given Esos' accounts

The Bank itself does not generate any messages, it only "answers" to received requests. In addition it does not keep any information about a server, which it communicates with. The Bank gives money to anyone, who is able to prove data ownership (i.e. who sends in a right time right MAC and SAuth).

12.2.2.4.2 Architecture of Bank

The Bank is the next Six in the Service. It extends Six with processing communication messages Bank-Client and Bank-Eso.



12.2.2.4.3 Implementation of Bank

12.2.2.4.3.1. Bank

The main class encapsulating whole functionality of the Bank. As each Six it runs all necessary threads and creates static object PayManager.

12.2.2.4.3.2. PaymentManager

It performs all operations united with payments. It provides communication with the Client and transfers its money to a newly created anonymous account.

12.2.2.4.4 Discussion, alternative solutions

12.2.2.4.4.1 Not unique ID of a payment in the Bank

One of the problems with payments is uniqueness of identification of the payment. While storing a file an Eso creates payment plan including for each payment its ID, date, SAuth and other aditional information. So inside the Eso

these "paymentID" are unique. But outside this Eso (namely in the Bank) could be different IDs the same. Than there is a problem when Eso requests for the payment. Here are several solutions:
- Requesting for the payment the Eso could send to the Bank also SAuth (for bigger security encrypted with the Bank's Public Key). SAuth is a string of random bytes and that is why here is very small probability, that they fits also.
- In next versions of the System - if confirmation of transfer of money sent by Bank to Eso will be incorporated - a new unique ID could be in this message.
- The Bank can answer with all suitable records, which it has. Then the Eso computes MACs for all these records and posts them back to the Bank.

12.2.2.4.4.2 Change of the Bank's Access Certificate

In present day we cannot store file for longer period than is the longest expire time of a Bank's Access Certificate. The reason is that storing data the Eso creates payment plan, in which individual payment is binded to a concrete Access Certificate of a Bank (precisely to its unique identifier). In that time the Eso could not know Access Certificates with longer "expire time". The solution is to bind the payment not to Access Certificate's ID but to the Bank itself. This would assume some unique identifications of Banks.

12.2.2.4.4.3 Retrieving money for a file, which has not been stored

A defect of present protocols in the Eternity Service is the fact, that when an user finds out that he has paid for data storage but the file is not accesible, he is not able to get his money back. Improving of these protocols, which would solve the problem, prepares Tonda Benes.

12.2.2.4.4.4 Conclusion and future work

In present version of the Service the Bank simulates all functions necessary for realization of real payment transactions. The connection to the system will be in each financial institution different.

12.3. System components interaction

12.3.1. Scenarios

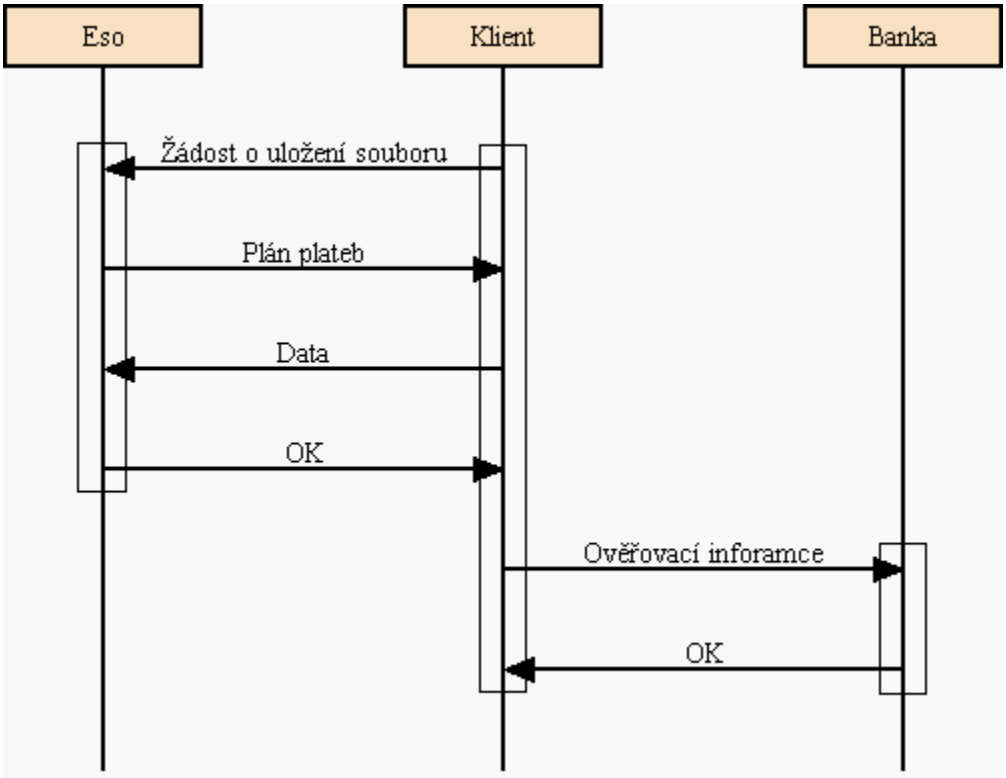
12.3.1.1. File storage scenario

Storage of a file to Eternity Service is done in two phases:
- request for storage
- file upload

User tells Client what file is to be stored, specifies keywords and number of Esos to upload to.

Client asks Mix for required number of Eso certificates. After their arrival Client generates a request for storage for each of them. Further we are concerned only about communication that is held between Mix and Eso.

This communication is described by the following picture:

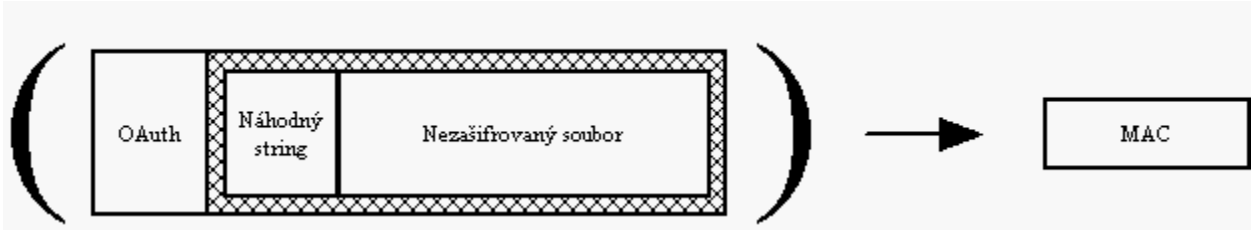


Client generates a request for storage. Eso that receives such a request tries to allocate requiered space on its disk. If succesful, it generates a payment plan as well, describing how much and when it will ask for money for the file storage. This payment plan is sent back to Client.

Current version of Client implicitly agrees with all payment plans it receives. A function that would show the payment plan to user to dis/agree should be implented in future.

If user accepts the payment plan, Client prepares file to upload. This procedure considers the requirement that Bank should be able to proove that Eso really stores the file it is asking money for. This is reached by concatenating a different random string to a file for each Eso the file is going to be stored to. The result is encrypted with Client's private key and this data are finally send to Eso along with Client's public key.

Before sending data to Eso there is generated one more random string (OAuth), which is added to encrypted data. MAC (Message Authentication Code) is counted from the result. Client stores all OAuth - MAC pairs.



When Eso receives the data to store, it checkes wether allocation took place before and saves the file. After that Eso sends a confirmation message to Client.

Now for each planned payment Client sends to Bank appropriate OAuth - MAC pair and waits until confirmation from Bank arrives.

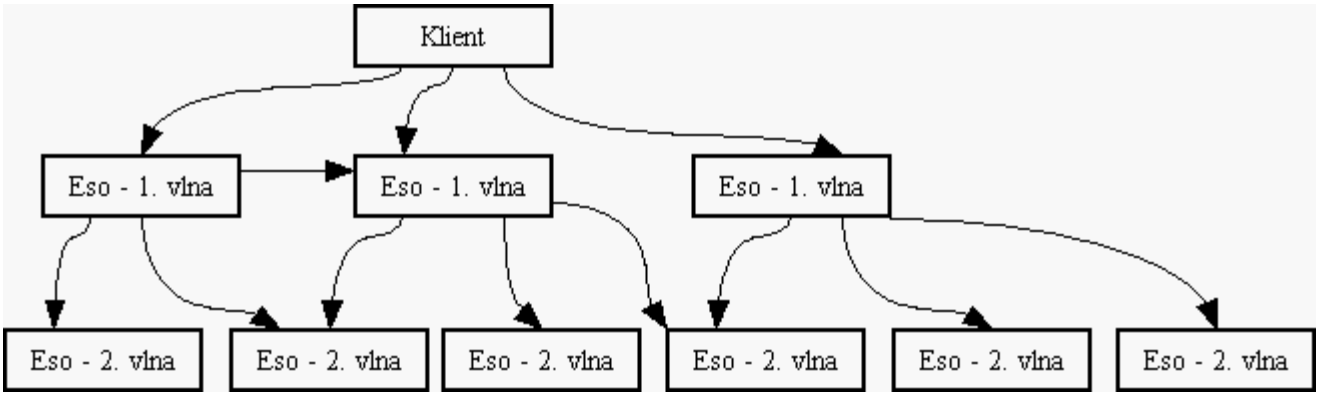
To explain why there is so much hacking around the file storage is not the

purpose of this document. Look for it somewhere else (try e.g. Tonda Benes - idea author).

12.3.1.2. File search and download scenarios

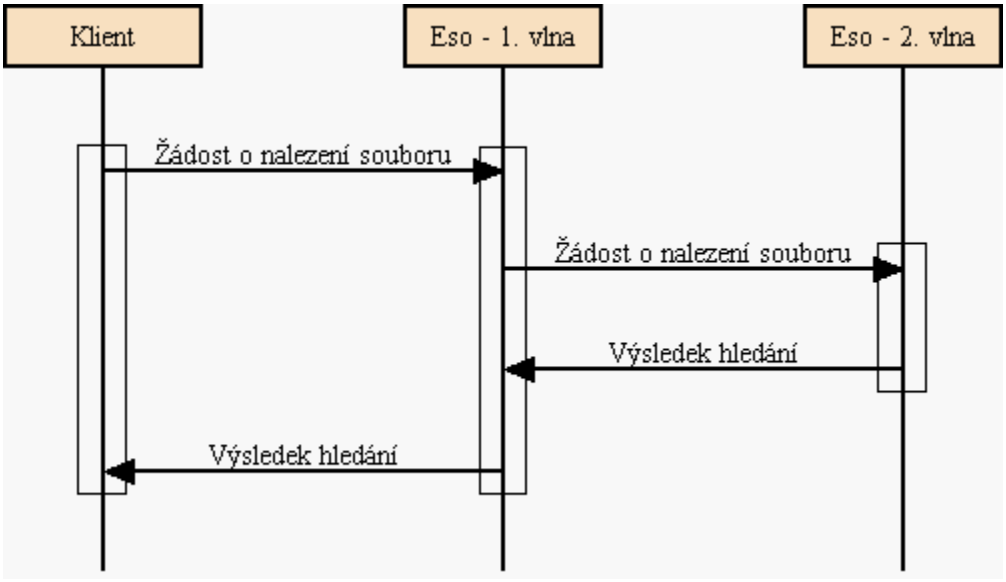
12.3.1.2.1. Search trees

Each file is uploaded into the Service together with a unique identificator (ffid) and some keywords. According to these pieces of information one can find and download the file. The access certificates of Eternity Servers change in time (because of expiration time - see section Access Certificates and Contacting the servers). Thus even the person, who uploaded the file may not know any of the valid certificates of Eso that stores the file. The search for file is done using the search tree (vertices are Esos). The depth and width of the search tree are specified by the user, but each Eso can lower these numbers by a constant specified in the config file. Configuring these parameters disables any attackers to fill up the system with many messages by specifying very large and deep search tree. The picture shows a search tree of depth 2.

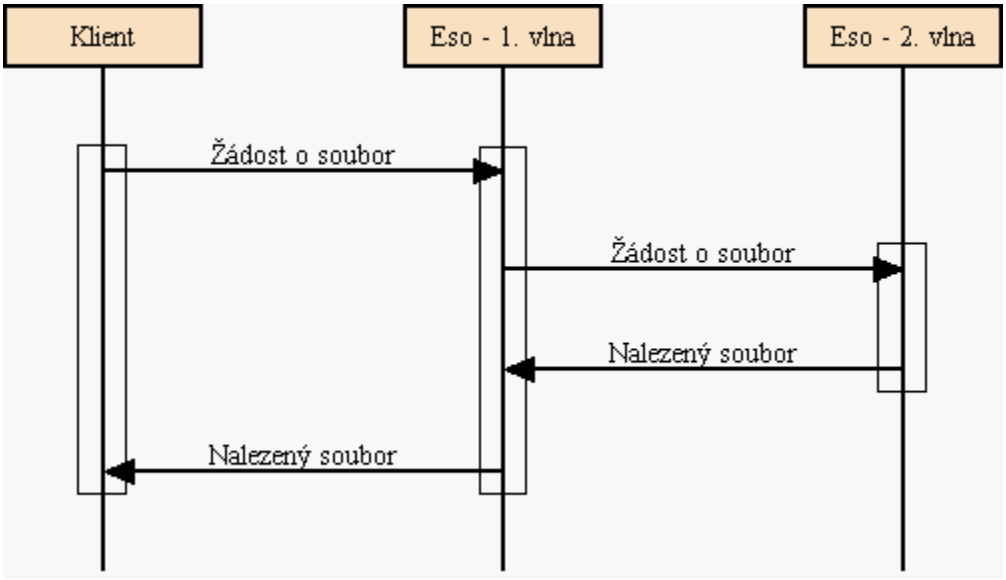


12.3.1.2.2. Search for file vs. download

We distinguish terms search and download. User, who does not know the unique identificator (ffid) of file first SEARCHES for file using its keywords. To prevent useless transports of large data, we proposed a protocol, in which the responses to request for search are just headers of found files with further information (now only the appproximate size) about the file and its ffid.



According to the ffid the user can send a request for downloading the file.



12.3.1.2.3. Forwarding and safety

Each Eso that is not a leaf of the search tree must store pieces of information about the Six (Client or Eso) that sent the request for download or search. These pieces of information are stored on the server only for the time necessary to search through the rest of the search tree. This method cannot be used in the contrary way (to save the information, where was the data found and thus make easier the download). The information "who sent the data" is much less critical than the information "who has the data". Storing the information about the owner is again good for the enemy, who is looking for the machine that owns the data. While downloading the file the search tree is thus created once more (nobody knows, where the requested file is) and there is a possibility, that in this case the file will not be found.

12.3.1.2.4. Cutting through the answers

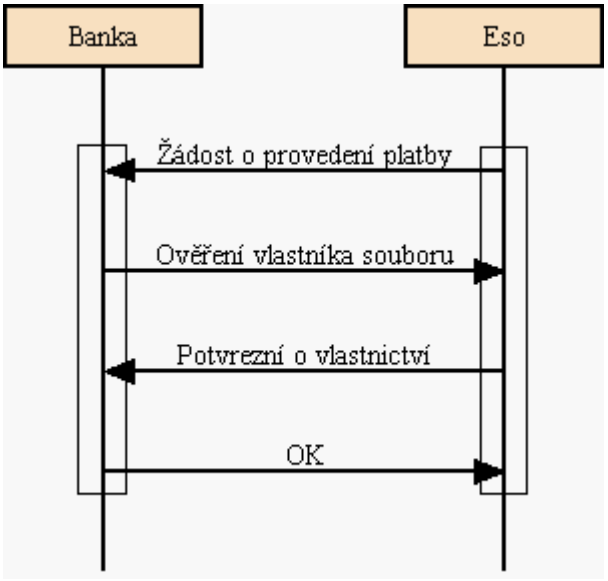
Each file can be stored in Service in several copies (all of them have the same ffid and keywords). The user doesn't care, which copy he gets. To

prevent the enemy to find out, how many copies there are in the Service, the answers to request for searchand download are on each non-leaf Eso cut through and the user gets only one of responses with the same ffid.

12.3.1.2.5. Summary

For search and download the search trees are used. Request for download must contain the unique indentificator of the file. For search keywords are needed. For each search or download new search tree is generated. The answers are on each non-leaf Eso cut through and user gets from one search tree only one answer about each found file.

12.3.1.3. Payment for stored data scenario



When the time for a payment for stored data comes up, the Eso contacts the correspondent Bank and identifies the payment.

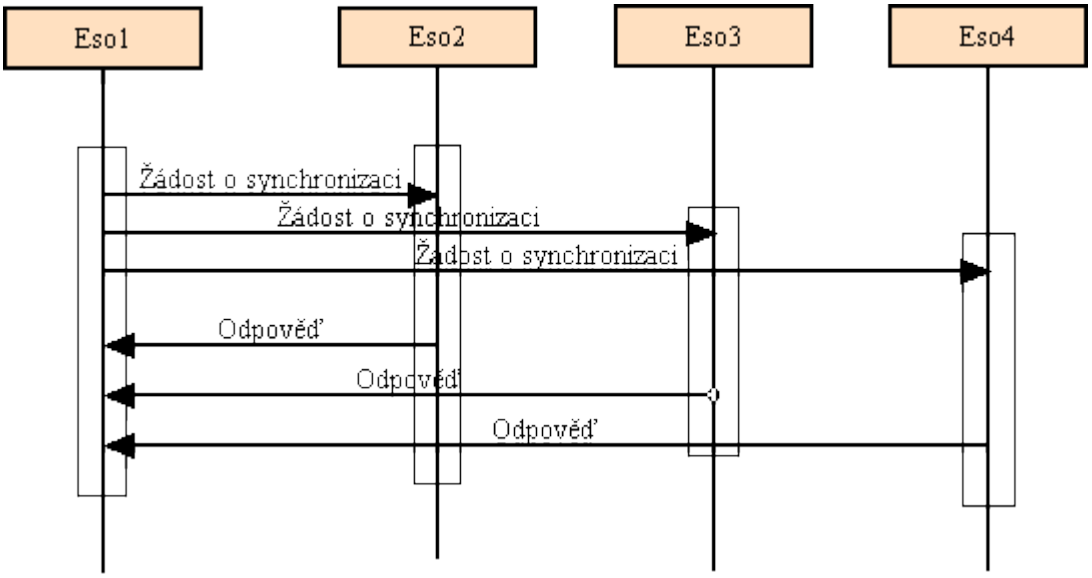
The Bank tries to search the record with given payment. If it has not succeeded, it sends to the Eso a negative answer. Such answer it sends also when the date in the record is not came up. Otherwise it challenges the Eso to compute MAC (by sending OAuth to it).

OAuth is secret key served for computing checksum (MAC), which is used for proving ownership of the file (see chapter 11.10). The Client sends this key to the Bank in payment plan, when storing his data.

Based on received OAuth and commited data the Eso compute MAC. This checksum it sends together with SAuth to the Bank.

The Bank compares its information with the arrival ones and if everything is OK, transfer the money from the anonymous account to Eso's accout.

12.3.1.4. Time synchronization scenario



Time synchronization starts when expired timeout of beginning of synchronization. Requests for time are send to random Esos, it is set new unique identification number for this synchronization, time of synchronizatin's beginning, timeout to synchronization's ending and timeout to new synchronization's beginning.

When some Eso gets request for time, this Eso adds the actual time to request message and sends it back to sender. Return address is included in request message.

The Eso which starts synchronization stores responses into the table. When number of responses is greater than number that we need, this Eso starts to compute divergence from system time. At the first responses are sorted by time, then extreme values are deleted and it is computed average from all values. If divergence is greater than is possible, information about it is written to logfile and in Scheduler state of system time is set as bad time.

When timeout of ending synchronization is expired, actual synchronization is break off and information about it is written to logfile.

Alternative calculation of divergence can be this formula:

divergence = ((t3 - t2) - (t2 - t1))/2

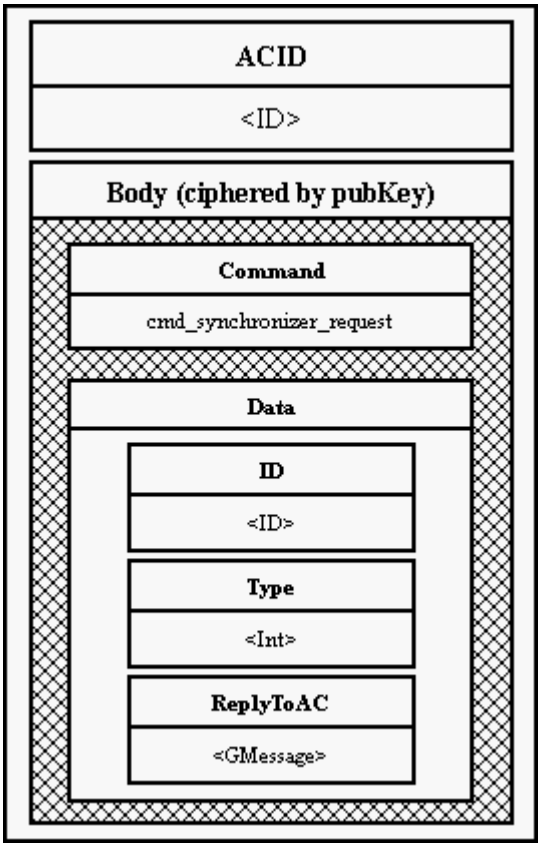
t1 ... beginning of synchronization
t2 ... time of getting request
t3 ... time of getting response

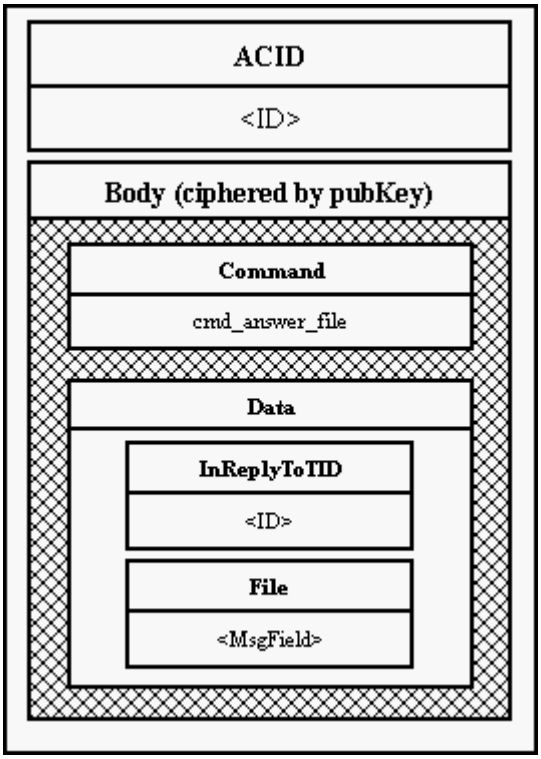
We assume that time to other Eso and back is almost the same.

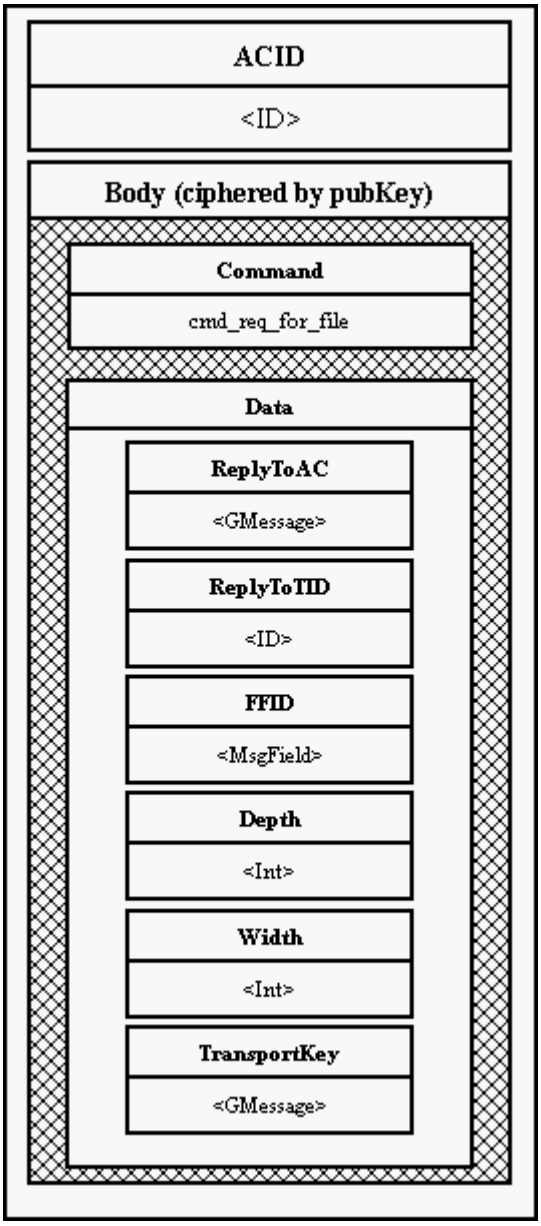
Other alternative is computing median from divergences than compute average. The median has better quality.

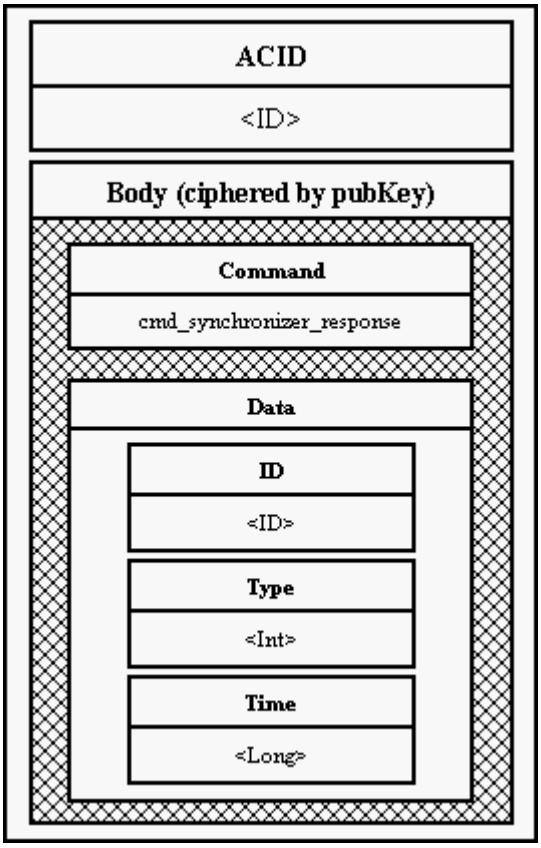
12.3.2. Protocols

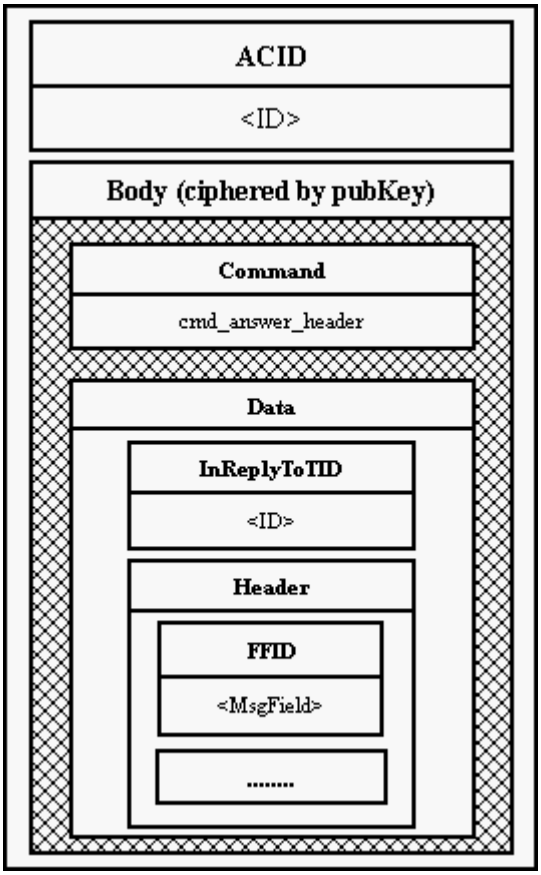
12.3.2.1. Six-Six

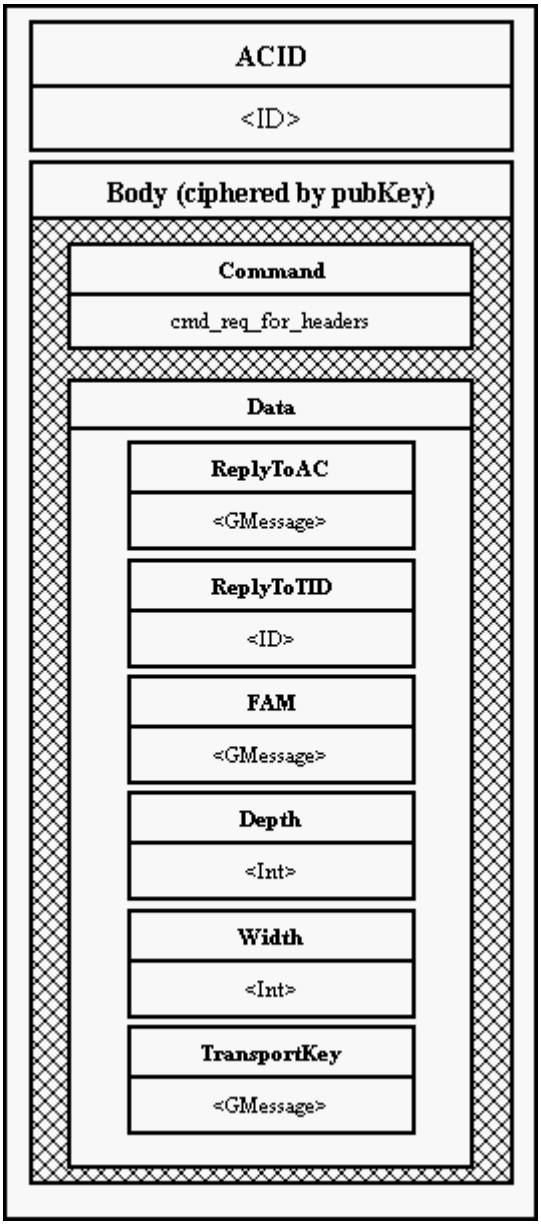

ACID
<ID>
Body (ciphered by pubKey)
Command
cmd_synchronizer_request
Data
ID
<ID>
Type
<Int>
ReplyToAC
<GMessage>

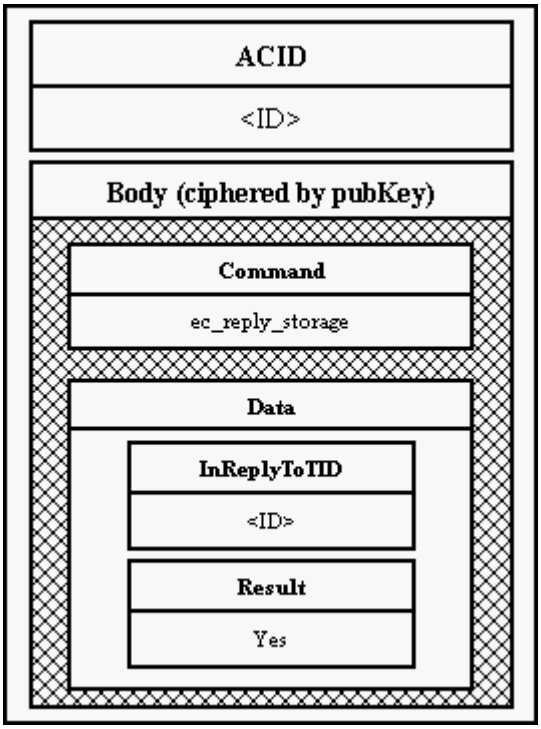

ACID
<ID>
Body (ciphered by pubKey)
Command
cmd_answer_file
Data
InReplyToTID
<ID>
File
<MsgField>

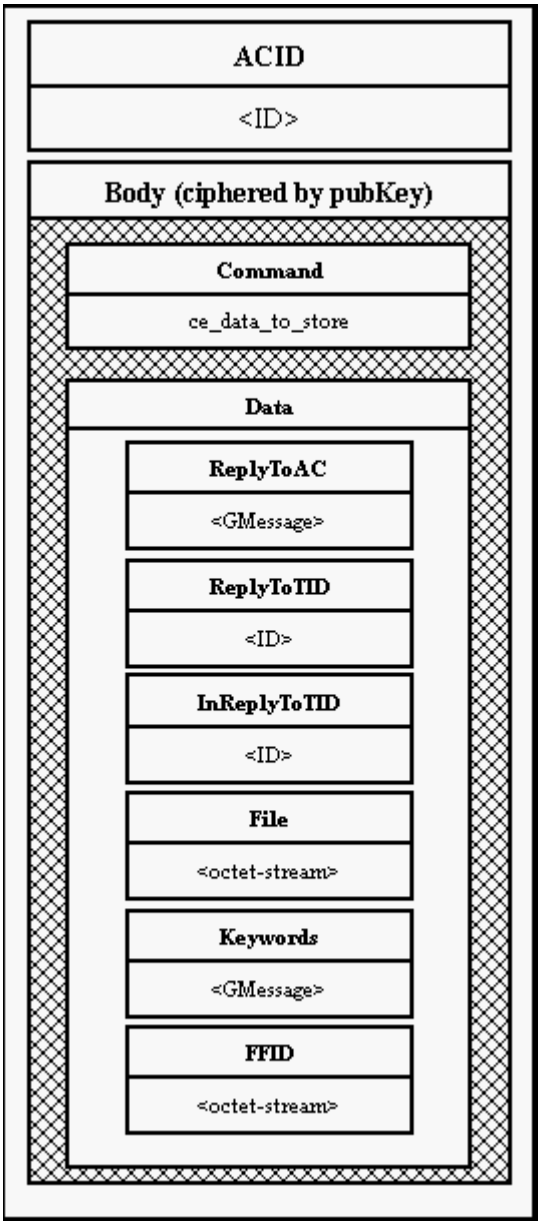

ACID
<ID>
Body (ciphered by pubKey)
Command
cmd_req_for_file
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
FFID
<MsgField>
Depth
<Int>
Width
<Int>
TransportKey
<GMessage>

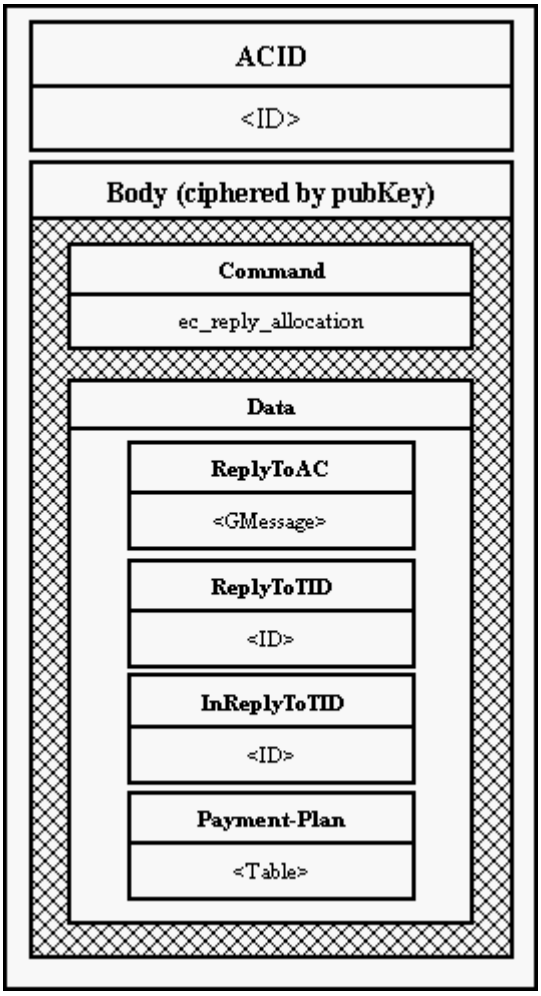

ACID
<ID>
Body (ciphered by pubKey)
Command
cmd_synchronizer_response
Data
ID
<ID>
Type
<Int>
Time
<Long>

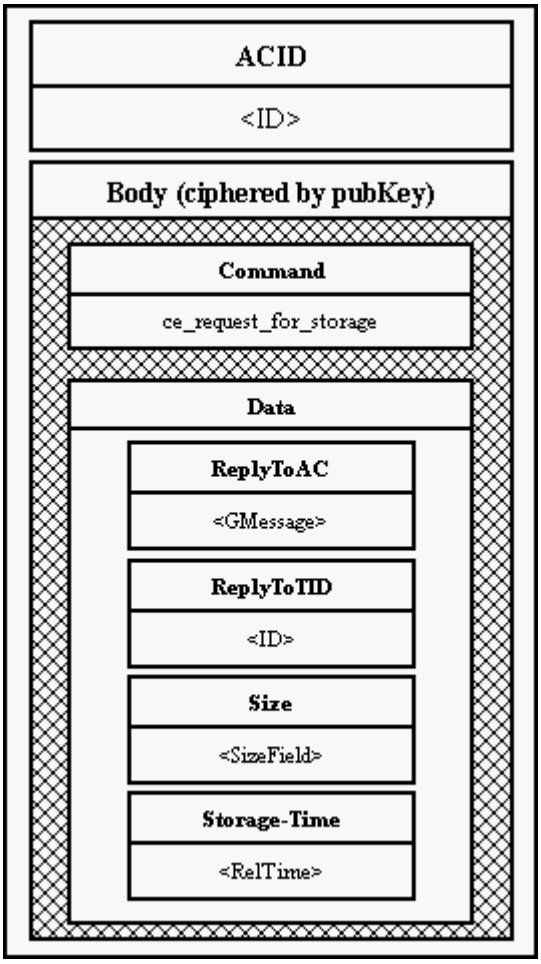

ACID
<ID>
Body (ciphered by pubKey)
Command
cmd_answer_header
Data
InReplyToTID
<ID>
Header
FFID
<MsgField>
.....

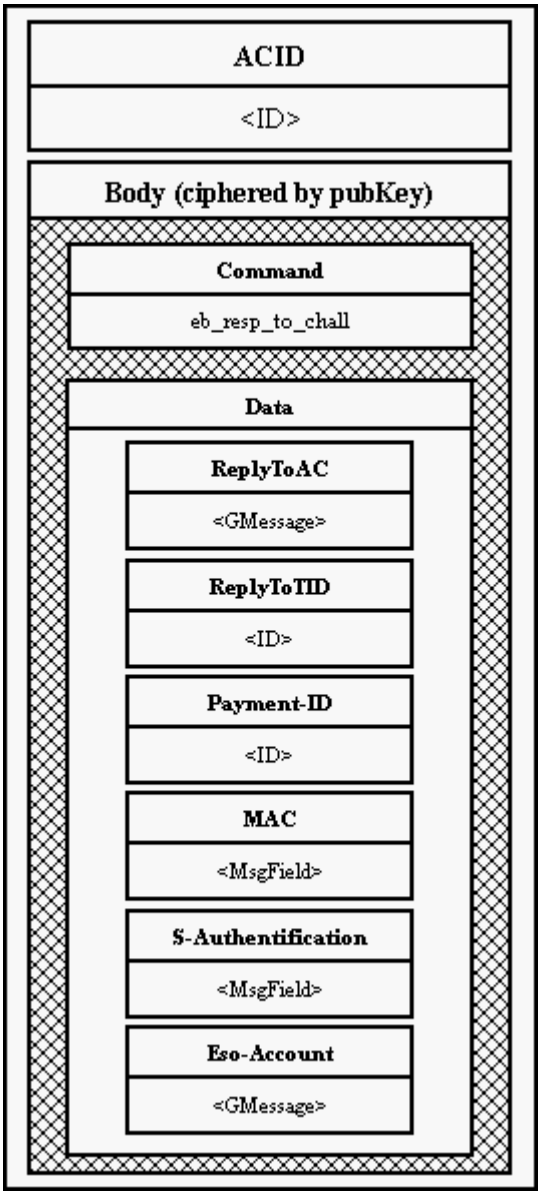

ACID
<ID>
Body (ciphered by pubKey)
Command
cmd_req_for_headers
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
FAM
<GMessage>
Depth
<Int>
Width
<Int>
TransportKey
<GMessage>

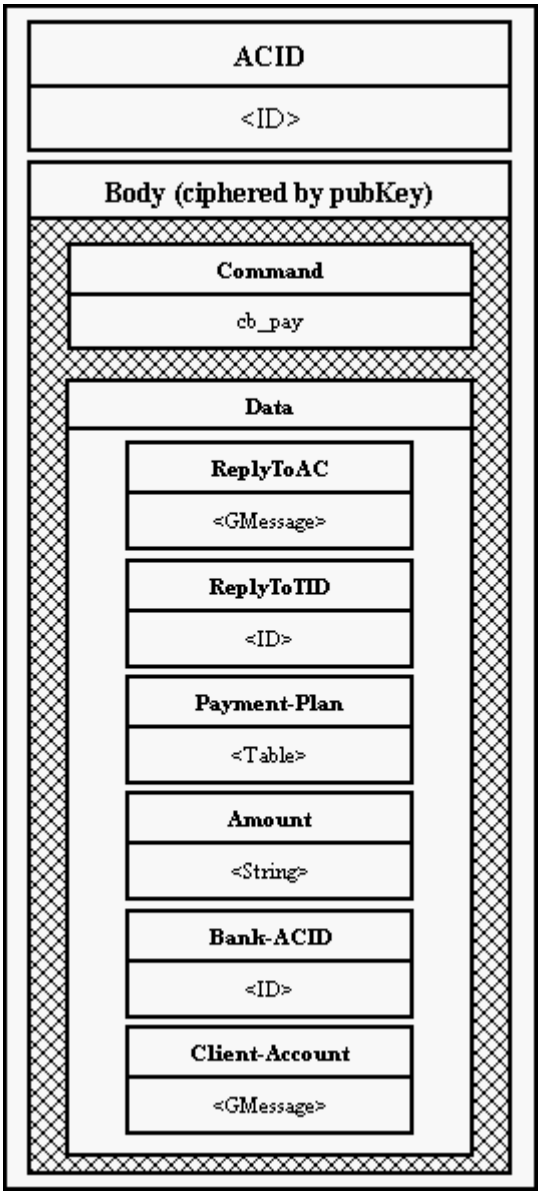

ACID
<ID>
Body (ciphered by pubKey)
Command
ec_reply_storage
Data
InReplyToTID
<ID>
Result
Yes

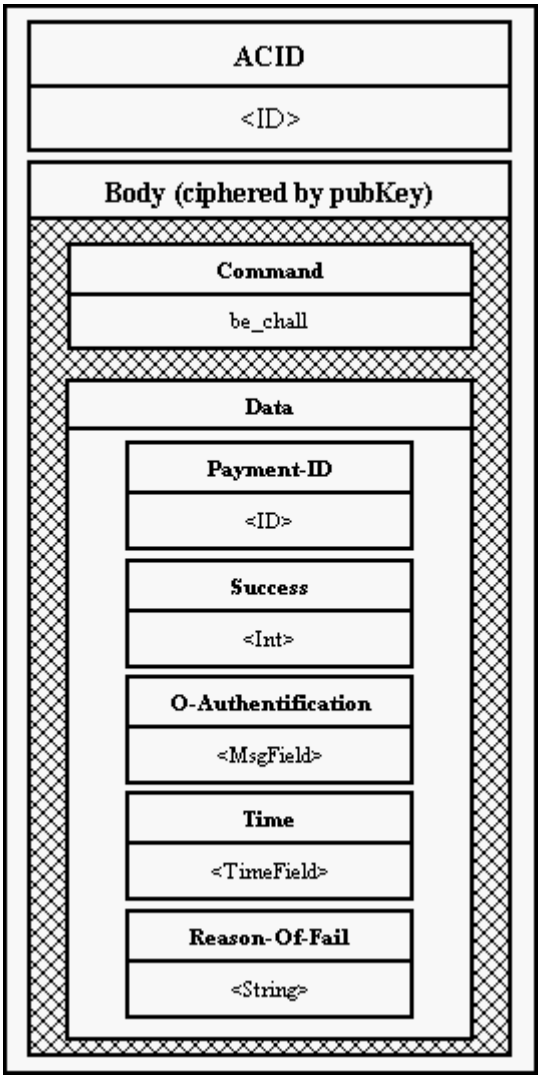

ACID
<ID>
Body (ciphered by pubKey)
Command
ce_data_to_store
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
InReplyToTID
<ID>
File
<octet-stream>
Keywords
<GMessage>
FFID
<octet-stream>

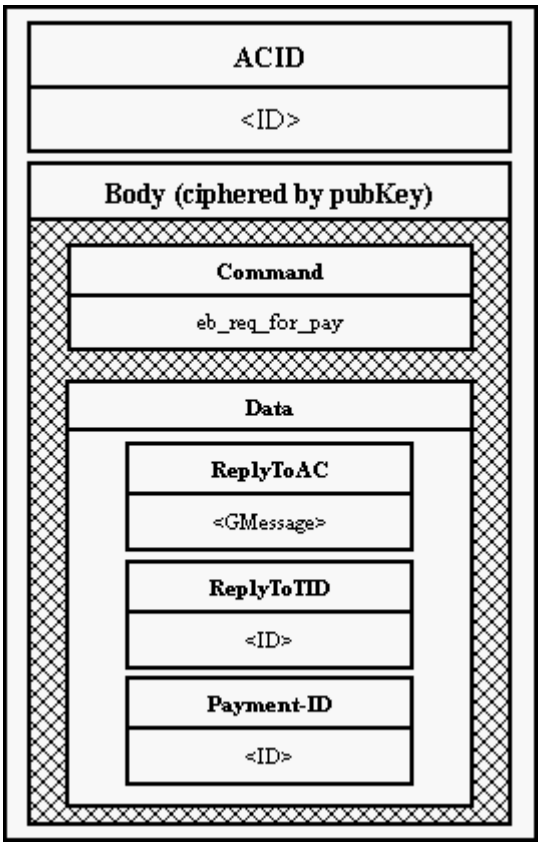

ACID
<ID>
Body (ciphered by pubKey)
Command
ec_reply_allocation
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
InReplyToTID
<ID>
Payment-Plan
<Table>

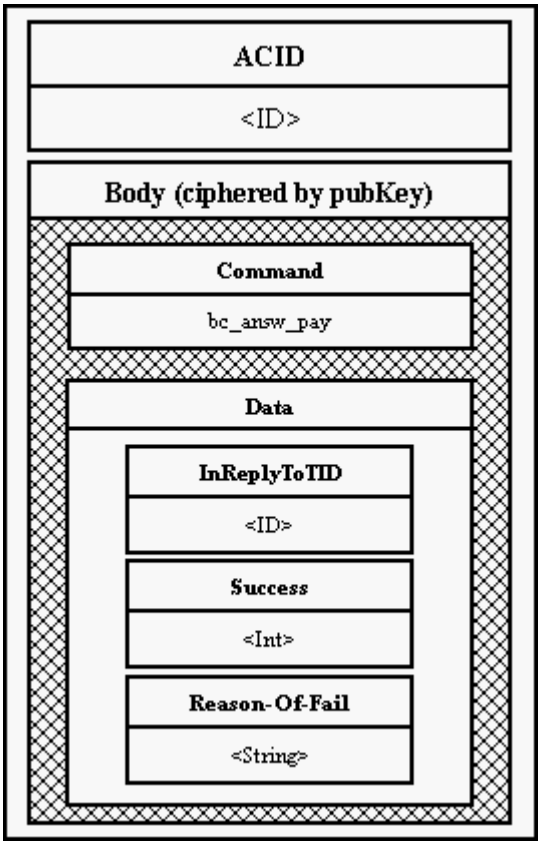

ACID
<ID>
Body (ciphered by pubKey)
Command
ce_request_for_storage
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
Size
<SizeField>
Storage-Time
<RelTime>

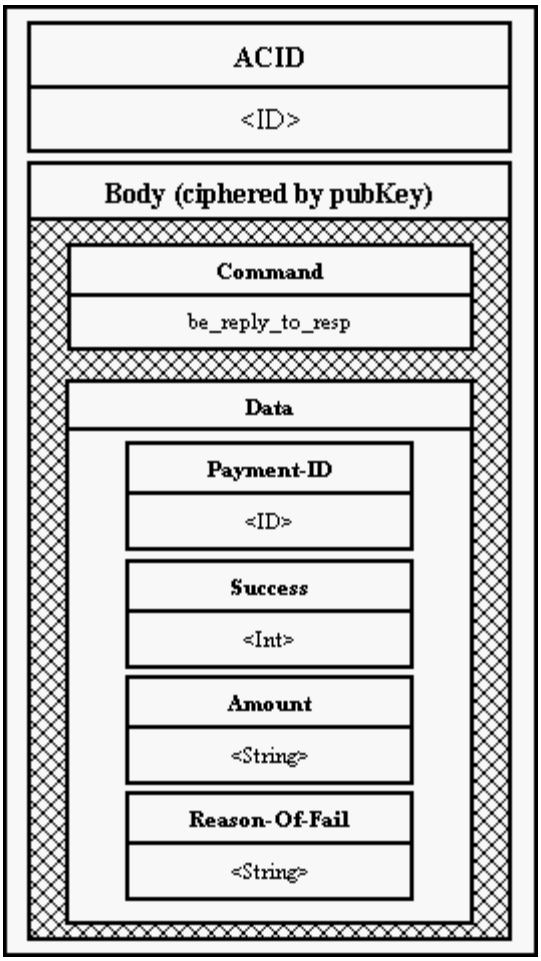

ACID
<ID>
Body (ciphered by pubKey)
Command
eb_resp_to_chall
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
Payment-ID
<ID>
MAC
<MsgField>
S-Authentification
<MsgField>
Eso-Account
<GMessage>


ACID
<ID>
Body (ciphered by pubKey)
Command
cb_pay
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
Payment-Plan
<Table>
Amount
<String>
Bank-ACID
<ID>
Client-Account
<GMessage>

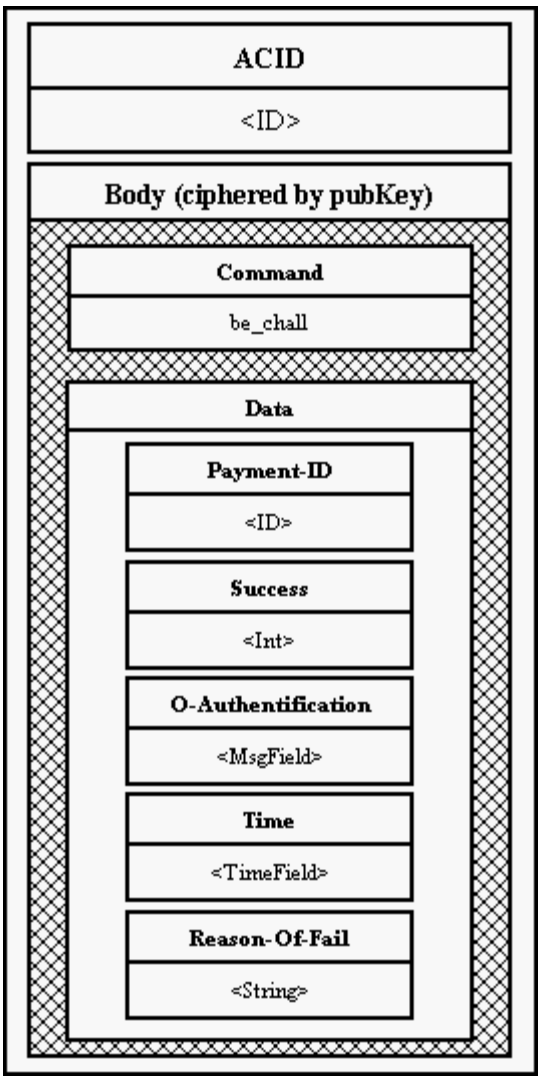

ACID
<ID>
Body (ciphered by pubKey)
Command
be_chall
Data
Payment-ID
<ID>
Success
<Int>
O-Authentification
<MsgField>
Time
<TimeField>
Reason-Of-Fail
<String>

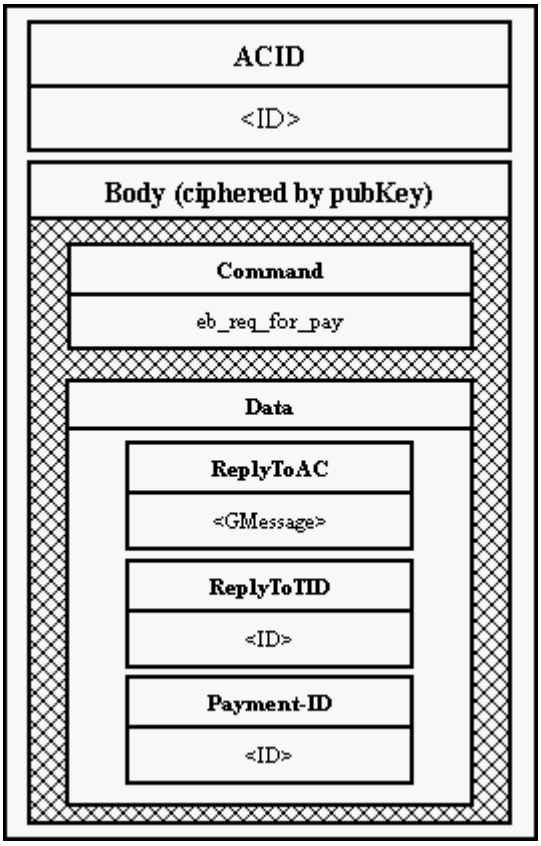

ACID
<ID>
Body (ciphered by pubKey)
Command
eb_req_for_pay
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
Payment-ID
<ID>

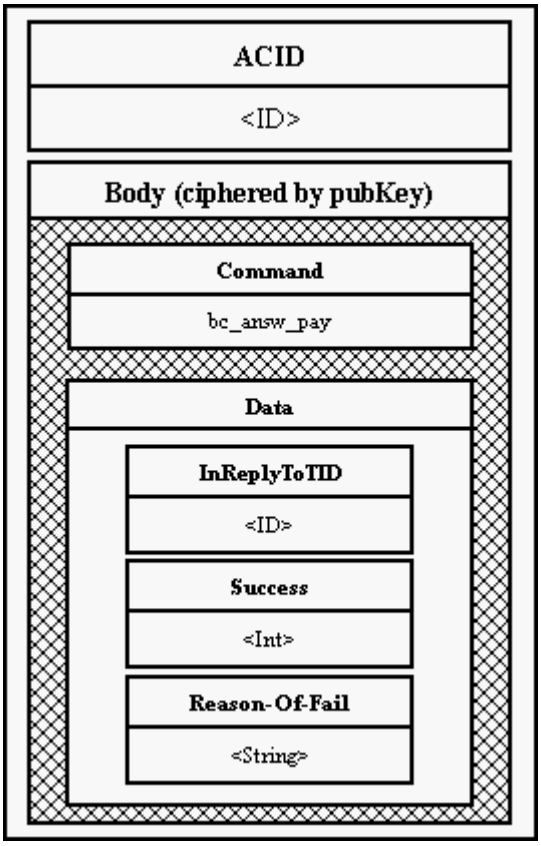

ACID
<ID>
Body (ciphered by pubKey)
Command
bc_answ_pay
Data
InReplyToTID
<ID>
Success
<Int>
Reason-Of-Fail
<String>

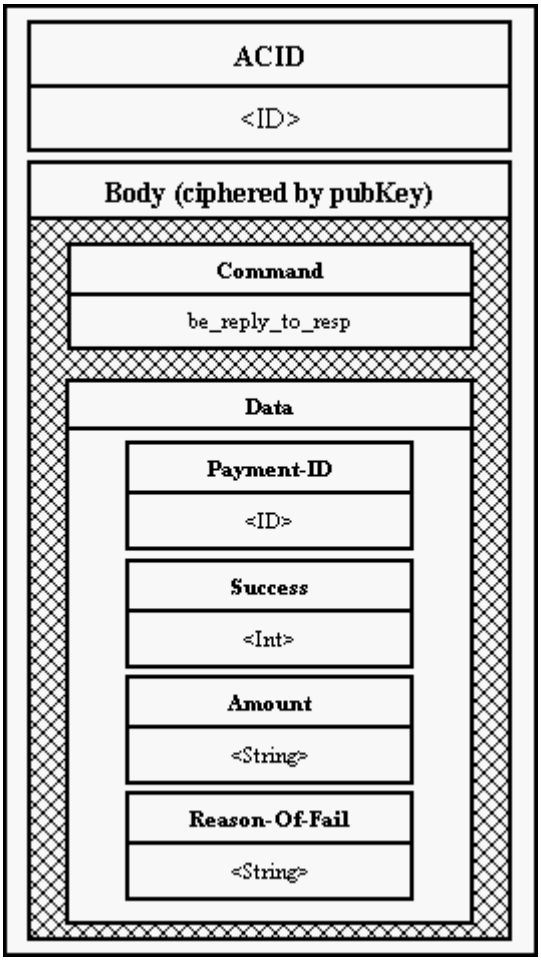

ACID
<ID>
Body (ciphered by pubKey)
Command
be_reply_to_resp
Data
Payment-ID
<ID>
Success
<Int>
Amount
<String>
Reason-Of-Fail
<String>

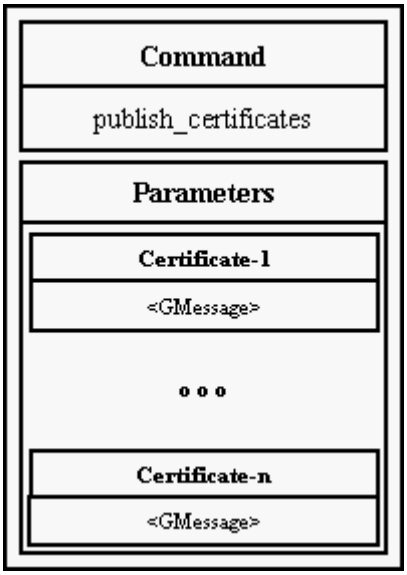
12.3.2.2. Six-Mix

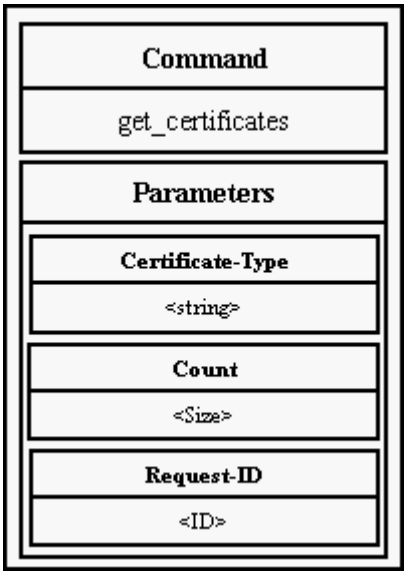


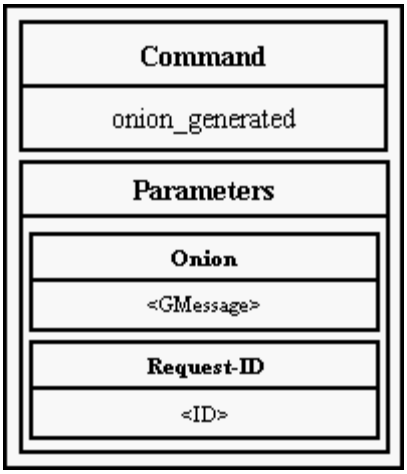

ACID
<ID>
Body (ciphered by pubKey)
Command
eb_req_for_pay
Data
ReplyToAC
<GMessage>
ReplyToTID
<ID>
Payment-ID
<ID>

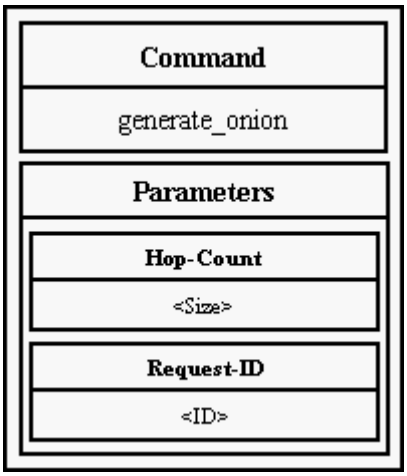

ACID
<ID>
Body (ciphered by pubKey)
Command
bc_answ_pay
Data
InReplyToTID
<ID>
Success
<Int>
Reason-Of-Fail
<String>

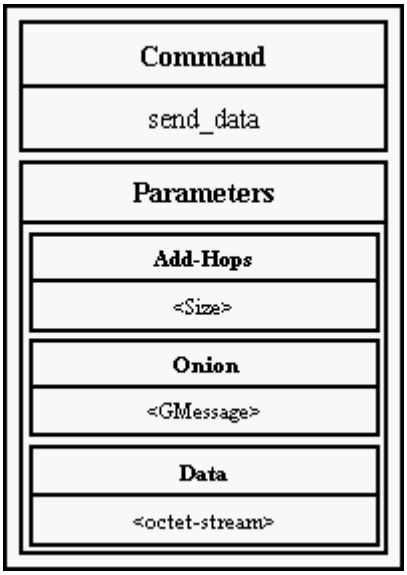

ACID
<ID>
Body (ciphered by pubKey)
Command
be_reply_to_resp
Data
Payment-ID
<ID>
Success
<Int>
Amount
<String>
Reason-Of-Fail
<String>


Command
publish_certificates
Parameters
Certificate-1
<GMessage>
o o o
Certificate-n
<GMessage>

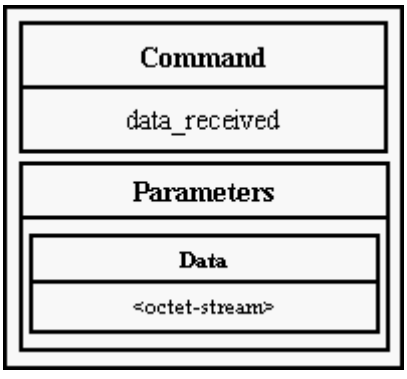

Command
get_certificates
Parameters
Certificate-Type
<string>
Count
<Size>
Request-ID
<ID>


Command
onion_generated
Parameters
Onion
<GMessage>
Request-ID
<ID>


Command
generate_onion
Parameters
Hop-Count
<Size>
Request-ID
<ID>


Command
send_data
Parameters
Add-Hops
<Size>
Onion
<GMessage>
Data
<octet-stream>


Command
certificates_received
Parameters
Request-ID
<ID>
Certificates
Certificate-1
<GMessage>
o o o
Certificate-(n-1)
<GMessage>
Certificate-(n)
<GMessage>


Command
data_received
Parameters
Data
<octet-stream>

12.4. Common objects

12.4.1. LogFile

Enables other objects to write debug messages, warnings and errors to log file.

12.4.2. ConfigFile

This class represents a configuration file and allows other objets to access information stored in it. Configuratio is stored in afile where lines beginning whith '#' and empty lines are ignored (considered to be comments).

First word on each line is considered to be key and the rest of line is the value. These pairs are then stored in memmory in a hash array for fast access (STL is used here). This array is protected by alock so save reload of config file should be possible.

All objects shoul not keep its copies of configuration data and should ask ConfigFile as this ensures data consistency.

12.4.3. Cipherer

Cipherer is a wrapper for RSARef toolkit. Its methods enable to process data of any length (ie. data given needn't be aligned to blocks of some bytes). The cipherer gets the data in the form of MsgField or GMessage, because almost all data are transmitted in one of these objects.

Methods of Cipherer can do:

- to encode data with asymetric algorithm (currently, combination of asymetric and symetric cryphtography is used, but it is transparent for the programmer)

- to decode that data

- to encode and decode data with symetric algorithm, no block alignment is needed

- MD5

- digital signature

- to encode and decode a symetric key (in the form of GMessage) with RSA

- special methods for asymetric coding, for onion creation and for adding new peels on the top of existing onion

---

12.4.4. Data alignment

During symetric encoding the data is padded to the block alignment and with information about the length (withou padding bytes) it is all encoded. During decoding, padding bytes are thrown away.

12.4.5. Asymetric coding

Encoding of big data would be very slow if we used only RSA algorithm. So we implement it as a combination of asym and sym algorithms. Firstly, a temporary symetric key is generated. The data is encoded with this symetric key. Afterwards this key is encoded with public key of the recipient. The encoded symetric key and encoded data are given together and can be sent over unsecure network.

Cipherer is used in Mix mainly by objects Translator and MessageCreator, but Cipherer is quite generic object, portable to another programs.

12.4.6. Receiver

Receiver is an object that accepts messages form the Net. He parses the message into the GMessage object and sends it then Translator. Parsing the data given is a simple check whether the data is in the form of GMessage. Bad data are thrown away and a note to the log is written.

When data is OK, Receiver adds field Origin into the built GMessage. This field then identifies the GMessage inside of the Mix.

Receiver listens to the port that is written is the config file. This port with IP address is included into the Mix certificate.

12.4.7. Sender

Gets messages consisted of the data part and than several records (currently 1 or 2) containing port and address where the message can be sent to. Sender takes all records in turn and tries to connect. When succesful, inserts the record into the data part (data part is GMessage too) and sends the data and than goes for waiting for the next message.

Sender is used for communication Mix-Mix or Mix-Six. Sender in Six adds Six's identification (Six's name written in config file for Six and Mix) into the message. This added identification is for Mix to identify where the message came from. No check is made, because we assume that Six and its local Mix are running on a controled network (typically the same secured machine). Some security can be added with firewall settings or so. This information mean that no encoding is used for communication between Mix-Six. If needed, use ssh tunneling, for example.

12.4.8. GMessage

GMessage is a flexible structure letting us work with messages in a simple way. Each GMessage holds several named fields (MsgField), which can keep data of any type. These fields provide a lot of conversion methods to simplify sending & receiving data of different types.

Nesting is one of the most important properties of GMessage which goes hand in hand with the layered architecture of Eternity Service.

All messages being sent in the system are created and parsed using GMessage.

12.4.9. MessageQueue

MessageQueue is a shit. Should be written with STL, but isn't. Do not ask us why.

12.4.10. Debugable

This class is the ancesstor of all objects that want to write into LogFile. It keeps a pointer to LogFile that is written to by default. Runnable is a child of this class. This class also encapsulates the setting of current level of debug messages that shoul be written to the log file. This level can be set for log file and then different one for particular objects.

It sholud be possible to set the debug level through config file. Now it is possible to set this level just at compile time. There is a syntax highliting file in the distribution that paints the logs with nice colors ;-)

12.4.11. Runable

It represents an object that runs as a standalone thread. This class enforces implementation of method Run() that is run in the thread. Some of the Runnable objects are Sender, Receiver, Translator, Linker, Majordomo's etc. All these objects are marked as ovals. The method Run of Runable object cannot be passed to pthread_create() directly so a wrapper function is used instead. This function just calls method Run of specified object.

12.4.12. RandomGenerator

An object that generates random bytes. RandomGenerator uses random() function that is seeded using data read from /dev/urandom. This object may be rewritten tousesomehardware randomgeneratorto generate "better" data.

There was a problem with reading /dev/urandom on FreeBSD 2.2.6 so we seed using time if reading fails - just for testing.

---

13. Summary

After one year of work on this project, we designed and implemented a functional framework of the Service. We implemented private and anonymous communication medium, that is realized by virtual net consisted of Mix servers. We designed an abstraction of a server that uses this virtual net and we made use of this abstraction for implementing of basic servers needed for functionality of the Service. These servers are Eternity server (Eso) that ensures higly reliable availibility of data that is stored on this server. Next, we implemented Access Certificate Server (ACS) that is a gate to the Service and ensures distribution of information about how to contact another servers. Pro user access to the Service, we implemented Client that enables to access to services provided by the Service. Except of this, we made a program that could become a core for software included into financial institutions that would want to support the Service (ie. provision of anonymous accounts).

---

14. Conclusion and Thanks

First of all, we would like to thank our project leader, Tonda Benes for cool idea of starting this project. Because of this, we got a perfect chance to learn a lot of new things, theoretically interesting and yet usable in a real computer life. FreeBSD system, unknown for us before, helped us to get more deeply into the fascinating world of Unix. Last, but not least, we would like to thank Ken Thompson for Unix philosophy, Dennis Ritchie for C, Bjarne Stroustrup for ++, UCLA for BSD and vi, Bram Moolenaar for m, FreeBSD Org for system, Microsoft Senior Recruiter Meggie Gougan for a nice password, Italians for spagetti and Metallica, Alanis Morissette and Rolling Stones for support.

---

15. Literature

[And] - R. J. Anderson - The Eternity Service, lesson published at Pragocrypt'96
[TB1] - A. Benes - The Eternity Service - A New Solution to the Data Storage and Recovery Problem? - preliminary study, unpublished
[TB2] - A. Benes - How To Achieve the Eternity, manuscript, published in Datasem 98 Proceedings, 1998
[Cha] - D. Chaum - Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms, Communications of the ACM, vol. 24, Feb. 1981
[Gold] - D. M. Goldschlag et.al. - Hiding Routing Information, Workshop on Information Hiding, Cambridge, UK, May 1996
[MCST] - RFC1301 - Multicast transport protocol
[Rand] - RFC1750 - Randomness recommendations for cryptography
[RSVP] - RFC 2205 - The resource reservation protocol
[X509] - CCITT Recommendation X.509, The Directory-Authentication Framework, Blue Book - Melbourne 1988
[Kal] Cryptographic Message Syntax by B. Kaliski
[Lef] Samuel J. Leffler - The Design and implementation of the 4.3 BSD Unix operating system
[POS] POSIX Thread Primer
[Sem] Semaphores for threads - C++ library
[Bee] Beej's Guide to Socket Programming
[4.4] 4.4 BSD IPC tutorial
[FBSD] Documents from the official FreeBSD site (www.freebsd.org)
[Man] Manual pages of FreeBSD operating system
[GNU] GNU manuals for gmake, gdb, gcc

==========